The design of a canonical database system (PRECI)

S. M. Deen, D. Nikodem and A. Vashishta

Department of Computing Science, University of Aberdeen, King's College, Old Aberdeen, Aberdeen AB9 U2B, UK

PRECI is based on a canonical data model potentially capable of supporting user views of other models—notably CODASYL and relational ones—through local schemas and appropriate data manipulation languages. The canonical global schema consists of normalised relations and is backed up by a storage schema and a data dictionary. The model is being implemented at Aberdeen University as a generalised database system, to be used primarily as a tool for research in databases, with a modular design approach so that future changes can be incorporated easily. The CODASYL and relational subschema facilities are currently being developed; a relational algebra to be used for DM commands from FORTRAN programs has been provided.

The storage and access strategy in PRECI is based on internal record identifiers (or surrogates) **created largely in entity-identifier order. Entity records can be accessed very quickly—both randomly and sequentially—by surrogates or entity identifiers, partly with the help of a novel indexing technique, called** hash tree, **which is based on data compression and hashing.**

The model provides maximal data independence through load-time and run-time binding. The local user may define access paths independently of the storage schema. Data items, record types and set types can be added to or deleted from the global schema; the storage schema can be restructured and data pages and indexes can be easily reorganised. Changes in one schema do not require recompilations of the unaffected schemas. The DBA has complete control over the storage and indexing strategy which he can manipulate to improve performance. Five levels of optimisation are provided to enhance execution efficiency and minimise memory usage. Certain integrity checks are also carried out during run-time.

(Received October 1979)

1. Introduction

It is nearly a decade since database technology began in earnest, and a number of models of widely different capabilities and user facilities have now established themselves as the major systems, the foremost among them are the CODASYL and relational models, the others being IMS, ADABAS, SYSTEM 2000 and TOTAL. The presence of so many different models invariably creates a serious problem of portability, compatibility and intermodel communication; therefore, it seems that a generalised model which can support the user view of all other models as local interfaces would be a useful facility. This would solve not only the aforementioned problem, but also enable the user, for instance a programmer, to retain his skills and use the system through an appropriate local interface as his familiar old model without requiring any major reorientation and without needing any substantial changes in his old programs. Above all, such a generalised model will provide the combined user-facilities of the local models it supports, and will thus offer greater flexibility and convenience to the users, which in a distributed environment will be particularly desirable. There is already a growing awareness among many experts of the need for a common framework providing such interfaces, at least for the CODASYL and relational models. These considerations motivated us to study the feasibility of such a generalised data model (Deen, 1977b) and finally prompted us to embark on the design of PRECI (prototype of a relational canonical data model with local interfaces—Deen, 1980).

PRECI is a generalised database system based on a canonical data model potentially capable of user views of all major models through appropriate local schemas (subschemas) and host languages (**Fig. 1**); but in the version under development only the CODASYL and relational facilities without any integrity and privacy constraints, are being implemented. The global schema is described in a canonical form using normalised relations (Fagin, 1977), and is backed by a storage schema and a data dictionary system. A stand-alone query-language facility is also contemplated. The design of PRECI has been partly dictated by the need for a generalised system, which can also be used as a test vehicle for research in the various aspects of databases including distributed databases. The structure of PRECI is modular and flexible with an open-ended approach so that changes can be incorporated easily.

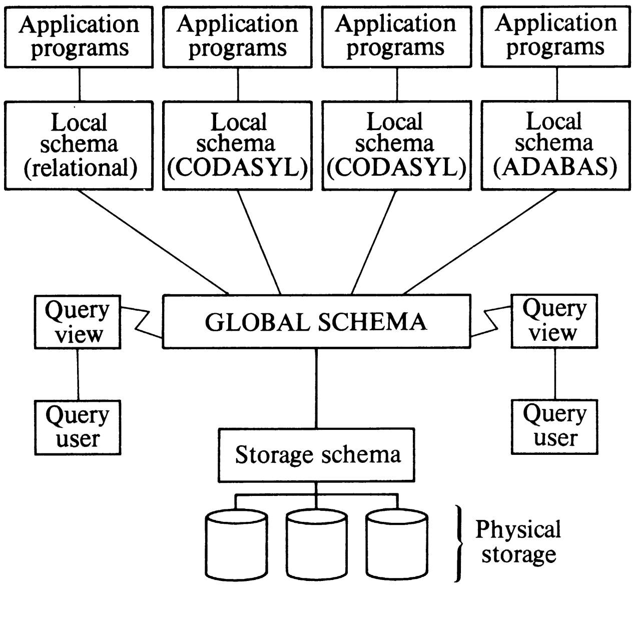


Fig. 1 Architecture of PRECI (there can be any number of local schemas for every local model, each local schema catering for an arbitrary number of application programs)

Run-time efficiency, minimal memory usage, optimisation facility, data independence and ease in restructuring and reorganisation are the major features of the model. The basic

CCC-0010-4620/81/0024-0200 $05.00

design phase is now complete. In this paper PRECI and the major features of its design will be described.

In Section 2 the canonical data model used in PRECI is outlined, followed by a discussion in Section 3 on local interfaces (particularly on the CODASYL and relational subschema) and by the basic PRECI commands and its relational algebra in Section 4. The next two sections are devoted to the storage system of PRECI, with Section 5 on storage structure and Section 6 on storage usage and efficiency. Associated issues such as data independence and ease of restructuring and reorganisation are covered in Section 7, and optimisation facilities in Section 8. The run-time control system is presented in Section 9, with a conclusion in Section 10.

2. Canonical data model

We define a canonical schema as the description of the inherent structure and usage constraints of data in a standardised form independent of its local interfaces. Local interfaces are the user views of other models supported as local schemas (subschemas).

Such a canonical schema, also referred to as the global schema in this paper, could describe the following objects:

entity records
entity relationships
usage constraints (privacy and integrity)

We intend to implement usage constraints in a later version, and hence these are not described here. In PRECI the content of the global schema is visible to the other schemas but the converse is not true since storage and local schemas are derived from the global schema. The storage schema provides specifications for:

storage descriptions of data
storage strategy (space allocation, placement and overflows)
access paths (keys, orderings and indexes)

To ensure data independence the entries in the storage and local schemas are kept invisible from each other. This allows storage (schema) entries to be changed without affecting the local views. In general, a local schema describes entity records and relationships, additional usage constraints (i.e. in addition to those provided in the global schema) and access paths. If these local access paths are supported by corresponding storage entries, access is faster. It is the responsibility of the database administrator (DBA) to examine usage statistics and to restructure the storage schema periodically.

As indicated earlier, normalised relations were used to represent entity records. This choice was mainly dictated by the brevity, elegance and convenience of the relational representation. Many other authors also seem to prefer a relational approach in data definition [see, among others, the papers by Adiba *et al.* and Benci *et al.* in Nijssen (1976); and Biller and Neuhold, and Schmid in Nijssen (1977)].

In PRECI each entity record of a relation is uniquely identified by an entity identifier (EID) made up of one or more attributes of the relation and concatenated in a preassigned sequence. An entity identifier is assumed to represent the entity and unique identifier of the entity record. The entity identifiers represent an essential characteristic of entity records, which should be declared in the global schema so that they are visible—through the local schemas—to the users who need them for the identification of records. Since by definition, an entity record is a collection of values describing the properties of an entity, we argue that the entity identifiers should in general be the primary means of access, and as such we require them to be declared as the principal access keys in the storage schema—referred to in PRECI as *primary keys*. Thus the EIDs specify both the inherent data characteristic (declared in the global schema) and the primary key (declared in the storage schema). This primary key determines the stored order of the entity records; and therefore the efficiency of the model crucially depends on the correct choice of entity identifiers, which must serve not only as unique identifiers but also as the principal key for direct and sequential access. Internally each entity identifier is represented by a unique internal number called a *surrogate* [see the paper by Hall *et al.* in Nijssen (1976)] which is not visible to the user and cannot be changed except by deleting the record and then reintroducing it as a new record.

PRECI allows two special attributes, called *time attributes* which can be declared in any relation. The *creation-time attribute* (CRE), if specified, will automatically record the date and time of insertion of each new tuple, which cannot be altered without deleting the tuple. The *update-time attribute* (UPD) notes the date and time of the last amendment of each tuple, and its value can be changed only by rewriting the tuple. The formats are *system defined* (SYS). These time attributes can be used for integrity and concurrency controls and also as keys in the storage or local schemas for time dependent ordering.

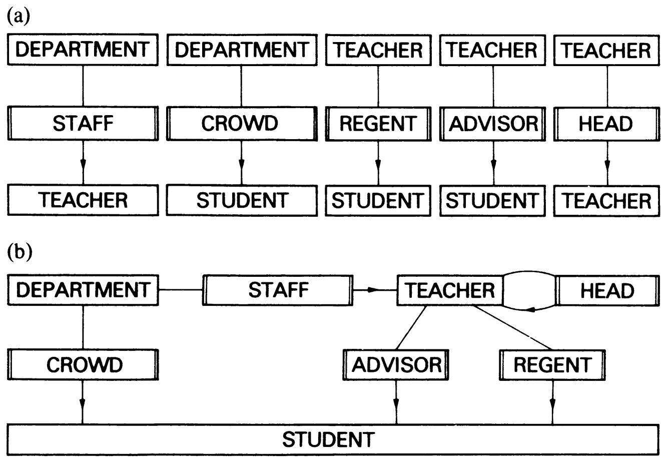
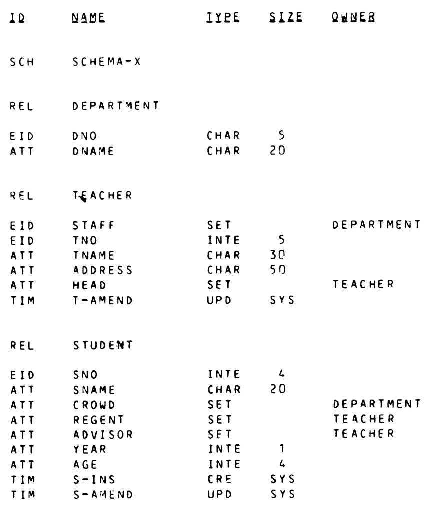


Fig. 2 **(a) Set types, and (b) relationships between set types (boxes with double vertical lines indicate set types)**



```
ID    NAME          TYPE   SIZE  OWNER

SCH   SCHEMA-X

REL   DEPARTMENT

EID   DNO           CHAR    5
ATT   DNAME         CHAR   20

REL   TEACHER

EID   STAFF         SET          DEPARTMENT
EID   TNO           INTE    5
ATT   TNAME         CHAR   30
ATT   ADDRESS       CHAR   50
ATT   HEAD          SET          TEACHER
TIM   T-AMEND       UPD    SYS

REL   STUDENT

EID   SNO           INTE    4
ATT   SNAME         CHAR   20
ATT   CROWD         SET          DEPARTMENT
ATT   REGENT        SET          TEACHER
ATT   ADVISOR       SET          TEACHER
ATT   YEAR          INTE    1
ATT   AGE           INTE    4
TIM   S-INS         CRE    SYS
TIM   S-AMEND       UPD    SYS
```

Fig. 3 **A canonical schema (sizes given here refer to the logical sizes only, the physical sizes are declared in the storage schema)**

As discussed in Deen (1980), 1:N relationships are represented as set types (basically a CODASYL term), essentially using the foreign key concept of the relational model. Consider for example the relationship among three entity record types TEACHER, STUDENT and DEPARTMENT represented by five set types STAFF, HEAD, CROWD, REGENT and ADVISOR as shown in **Fig. 2.** In some universities each student is assigned to two teachers, a regent and an advisor, the regent acting as the academic guardian in all matters except advice on courses which is given by the advisor. Teachers and students belong to departments, and it is assumed that, like a teacher, a student also belongs to only one department (the department of his/her honours subject) which is not necessarily the department of his/her regent or adviser. The teachers are also related to the head of the department who is another teacher. These five set types are represented for PRECI in **Fig. 3.**

In our schema representation, *set names* are replaced by *set attribute names*, each set attribute containing the entity identifier of the owner. Assuming that TNOs are unique only within departments, the EID of relation TEACHER is ⟨STAFF⟩ ⟨TNO⟩, that is ⟨DNO⟩ ⟨TNO⟩, and hence teachers are the automatic fixed members (in the CODASYL term) of the set type STAFF which Kay (1975) describes as a 'dependent' set type.

In relation TEACHER the identity of two separate attributes, STAFF and TNO making the EID is maintained. This is not so in set attributes REGENT and ADVISOR, each of which refers to the full EID of teachers as a single concatenated attribute constructed from DNO and TNO. This prevents the student from automatically becoming a member of his/her regent's or advisor's departments in addition to his/her own and thus avoids the confusion which would otherwise ensue. To represent M:N relationships in PRECI, one must use the link record approach [see Chapter 3 of Deen (1977a)]. Note that an update-time attribute has been declared in relation TEACHER; and both time attributes in relation STUDENT.

3. Interfaces

The canonical schema described above appears to be capable of supporting the user views of most models as local interfaces. We have studied CODASYL, relational and ADABAS models (Deen, 1980). In particular the feasibility of the CODASYL model was investigated with respect to the representation of (*a*) records and sets (*b*) set membership class (*c*) set selection criteria (*d*) record keys and search keys (*e*) set order criteria and keys; and (*f*) subschema entries.

The study shows that PRECI is not only capable of providing and sustaining a CODASYL subschema, it would also help remove inconsistencies by imposing certain constraints. The model gives greater data independence by giving the user freedom to define any key, set order criteria or set selection criteria in the local schema irrespective of the specifications in the storage schema. Time attributes in the global schema can be used to define the set order FIRST or LAST of the CODASYL model. PRECI interprets set membership criteria essentially as integrity constraints. We have also reached a positive conclusion for the ADABAS interface. Since the canonical schema is defined in terms of relations, the relational interface is naturally straightforward, except that PRECI does not allow the relational user to violate usage constraints such as the deletion of an owner record in the presence of member records. We describe below briefly the CODASYL and relational interfaces.

CODASYL interface

Most CODASYL entries (CODASYL, 1978) are unaffected by our model, except for minor changes to the alias section

```
MAPPING DIVISION.

ALIAS SECTION.

DN CROWD BECOMES POP.
SN CROWD BECOMES MOB.
DN STAFF BECOMES DEPT.


STRUCTURE DIVISION.

SET SECTION.

SD STAFF
   SET SELECTION THRU DATA-BASE-KEY.

SD REGENT
   SET SELECTION THRU CURRENT OF SET.

SD MOB
   SET SELECTION THRU DNO VALUE OF POP.


RECORD SECTION.

01  STUDENT.
     02 SNO              PIC 9(4) COMP.
     02 POP              PIC X(5).
     02 SNAME            PIC X(20).

01 TEACHER.
     02 DEPT             PIC X(5).
     02 TNO              PIC 9(5).

01 DEPARTMENT.
     02 DNO              PIC X(5).
     02 DNAME            PIC X(20).
```

Fig. 4 A CODASYL subschema (title division and realm section are not shown)

```
ID     NAME       TYPE    SIZE  HREL          ALIAS

REL    TEACHER                                TCR

PKEY   DNO        FKEY          DEPARTMENT
PKEY   TNO        INTE     5
ATT    TNAME      CHAR    30

REL    STUDENT                                STD

PKEY   SNO        INTE     6
ATT    SNAME      CHAR    30
ATT    REGENT     FKEY          TEACHER       RGT
ATT    YEAR       INTE     1
ATT    ADVISOR    FKEY          TEACHER       ADV
ATT    AGE        INTE     2
```

Fig. 5 A section of a relational subschema

and set selection criteria. To avoid confusion between a set name and the possible use of the same set name as a data item name—and also to improve clarity—we have introduced three new entries in the alias section, DN for *data item name*, SN for *set name* and RN for *record name*. Note that the system does not need these new entries to distinguish between a set name and data item name, but they are helpful to the user. A sample subschema is presented in **Fig. 4.** The set selection clauses used there are drawn basically from the provision of the CODASYL subschema facility (with reduced verbosity); but the database key (that is surrogate of the owner) option which we have retained as a useful facility is no longer available in the CODASYL model.

PRECI supports all the variations permitted in the CODASYL model between schema and subschema except that each subschema record is required to contain the EID of the global record. In addition, the application programmer is free to define any record key for record selection format 1 and 2 and set order criteria for format 4 as necessary (the CODASYL model permits only search keys for format 7 to be so defined —Deen, 1980). However, the present version of PRECI does not allow record types with repeating groups or multi-membered set types in the CODASYL subschema.

Relational interface

The relational subschema is a logical subset of the canonical schema with the following permitted variations:

(*a*) format of an attribute can be different and attributes can be presented in a different order;

(*b*) a local relation need not have all the attributes of the global relation, but the EID must be included in the local schema;

(*c*) the user may declare only those relations that are needed;

(*d*) attributes and relations can be renamed (alias column);

(*e*) additional usage constraints can be imposed (but not yet specified).

A sample relational schema is given in **Fig. 5.**

4. Languages

PRECI supports CODASYL DM commands and a relational algebra for FORTRAN programs. Here, only the relational algebra is described (Szymanski, 1978); the operations supported in order of decreasing precedence are given in **Table 1.**

Table 1

Conditions for selection	
Comparison operators	= <> < > ⩽ ⩾
Boolean operator	NOT
Boolean operator	OR
Boolean operator	AND
selection	:
division	/
join	*
projection	£
intersection	!
union and difference	+ and −

Commands at the same level of precedence are evaluated from left to right unless interrupted by brackets.

The particular precedence sequence in Table 1 encourages the formulation of relational expressions in an optimal manner in terms of execution efficiency. The syntax chosen yields compact expressions without unnecessary brackets and permits queries to be formulated either as a series of simple expressions (for beginners) or as a single complex expression (for the more experienced) as can be seen from the examples given below. In these examples domains are identified by attribute names. In the equi-join operation the common domain (possibly composite) is enclosed within brackets e.g.

```
R3 := R1(A1) * (B1)R2
R4 := R1(A1,A2,A3) * (B1,B2)R2
```

In the first case, the equi-join of R1 and R2—with common domains A1 and B1 respectively—is to be evaluated, the result being stored in R3. In the second case the common domain is the concatenation of three domains A1, A2 and A3 for R1 and of two domains B1 and B2 for R2. In division, the dividend relation must be effectively binary, and the divisor relation unary, through concatenations if necessary; the uncommon and common domains being separated by | in the dividend relation, e.g.

```
R3 := R1(A1|A2) / (B1)R2
R4 := R1(A1,A2|A6,A7,A8) / (B5,B2)R2
```

Any other domains of relations R1 and R2 will be ignored in the evaluation of the division. Attribute names and relation names can be redefined; and redefinition can be used to concatenate, e.g.

```
A == R(A1)
B == R(A1,A2)
```

A is an alternative name for attribute A1, and B for the concatenated attribute ⟨A1⟩⟨A2⟩ of relation R. The original names A1 and A2 can also be used along with A and B. In the examples given below it is assumed that a relation SCR with attributes *course number* (CNM), *student number* (SNM) and *examination marks* (MK) also exists in the subschema of Fig. 5.

(1) Extract the names of the first year students older than 20

```
SNAME£STD : YEAR = 1 AND AGE > 20
```

(the £ symbol can be read as *of* or *from* and the : symbol as *where* or *such that.*)

(2) Find the name of Bob's regent

```
PK == TCR(DNO,TNO)
RG := RGT£STD : SNAME = "BOB"

TNAME£TCR : PK = RG
```

Alternatively, in a single step

```
TNAME£TCR : (DNO,TNO) = RGT£STD : SNAME=
"BOB"
```

(where RGT£STD : SNAME="BOB" is a condition which includes another condition SNAME="BOB")

(3) Find the names of all students who took all the other courses

```
ALLIST := SCR(SNM|CNM) / (CNM)SCR
SNAME£STD(SNO) * (SNM)ALLIST
```

Alternatively

```
SNAME£STD(SNO) * (SNM)SCR(SNM|CNM) /
(CNM) SCR
```

(4) Find the names and marks of all first year students who scored more than 60 in course C1

```
(SNAME,MK) £ STD(SNO) * (SNM)SCR:CNM = "C1"
AND MK > 60
```

(The selection on SCR will be carried out before the join is evaluated.)

Other commands

Other important commands available in the relational interface are:

HOLDS *i*	to hold the surrogate of a selected tuple in position *i* in the surrogate buffer
RELE *i*	to free position *i*
GETS *i*	to get a tuple directly by surrogate (retained in position *i* by a HOLDS command)
GET{F/N/P}	to get the first, next or prior tuple in surrogate order
REPL	to store (replace) an amended tuple (Note that a primary key cannot be replaced)
DELE	to delete a tuple
INSE	to insert a new tuple
OPEN *rel.*	to open a named relation
CLOSE *rel.*	to close a named relation

Commands for accessing records sequentially in primary and other key orders will be implemented in the future. Note that surrogates are not visible to users, but can be used in the same run-unit for direct access if retained earlier by a HOLDS command. Integrity commands such as CHECKPOINT, ROLLBACK etc. are also supported.

5. Storage structure

As indicated earlier the storage schema basically contains the specifications for data storage and access paths. For each record type the user is required to specify the following entries:

(1) record description (including data items and their physical size);

```
RECORD SECTION

COPY ALL        (to copy the global schema description of
                records for the storage schema)

KEY SECTION
 ID       RELNAME        ORDER     CONSTITUENT-DOMAINS

 PKEY     DEPARTMENT     ASC       }
 PKEY     TEACHER        ASC       }  not needed for primary keys
 PKEY     STUDENT        ASC       }

AREA SECTION
 ID        NAME            AREA-NAME      PAGES      UNIT

 REL       STUDENT         ST-AREA        1000         1
 REL       TEACHER         TC-AREA         150         1
 REL       DEPARTMENT      DT-AREA           5         1

 PINDEX    STUDENT         PAREA1           25         1
 PINDEX    TEACHER         PAREA2           10         1
 PINDEX    DEPARTMENT      PAREA3            1         1

 SET       CROWD           SAREA1            5         1
 SET       REGENT          SAREA2            5         1
 SET       ADVISOR         SAREA3            5         1
 SET       STAFF           SAREA4            3         1
 SET       HEAD            SAREA5            1         1

SURROGATE SECTION
 RELNAME      OPTION  WH   WO   HCODE CCODE IPD     NHS     NOS

 STUDENT        1     20   10   DEF   DEF    70     700     500
 TEACHER        1     10    5   DEF   DEF    80     100     100
 DEPARTMENT     1     30   10   DEF   DEF   100     100     100

PINDEX SECTION
 RELNAME      OPTION  WH   WO   HCODE CCODE IPD     CKSIZE LOV GOV

 STUDENT      H-TREE  10    5    DEF   DEF   70       4     25  30
 TEACHER      H-TREE  10    5    DEF   DEF   75       4     20  30
 DEPARTMENT   H-TREE  10    5    DEF   DEF   80       4     20  25

 [If no set keys are specified a system-default option
 for order is invoked in the key section.]
```

Fig. 6 A section of a storage schema

(2) specification of the primary, secondary and set keys along with index descriptions;

(3) data areas with page size and overflow strategy for entity records;

(4) surrogate allocation strategy specified in surrogate directory;

(5) index areas with page size and overflow strategy separately for each of the following indexes: (*a*) PINDEX area for the primary key (see EID). Besides PINDEX, this area also holds as part of the header the general information on the relation and an optional surrogate directory (see later); (*b*) set areas for set indexes; (*c*) key areas for the indexes of secondary keys (only if secondary keys are specified);

(6) hashing technique including a key compression code for the surrogates, primary keys and other keys (if relevant).

A sample storage schema is shown in **Fig. 6** where

UNIT	defines the page size in terms of the basic operating-system unit of data transfer
HCODE/CCODE	hash/compression code for keys
WH/WO	home/overflow hash width
DEF	system default option for hash codes and compression codes
NHS/NOS	number of home/overflow slots.
IPD	initial population density of home slots
CKSIZE	compressed key size in bytes
LOV	percentage of space reserved in each home page of PINDEX for local overflows
GOV	percentage of PINDEX pages reserved for global overflows

For many entries default and alternative options exist. In cases of alternatives, guidance as regards their effectiveness is also given to the user. It is possible for records of different types to share the same data area.

A surrogate is constructed by concatenating an internal relation number and an effective key obtained by applying a suitable hashing algorithm on the primary key. Effective keys are derived in such a way that they largely maintain the primary key sequence (Deen, to be published). The hashing technique gives a set of hash values or hash slots—each for a group of tuples—with an average distribution W (hash width). The effective keys are assigned sequentially within a hash slot, all keys with hash slot I having the effective keys in the range [(I − 1)∗W] + 1 to I∗W. If the position of a tuple in a hash slot I is P, then its effective key is [(I − 1)∗W] + P (see **Fig. 7**). It is possible to restrict the initial allocation of surrogates to only a defined percentage of the hash width. Overflow hash slots are also used.

To minimise the standard deviation of W, the primary keys are compressed before hashing is applied. The compression algorithm used produces a more uniform distribution but retains the primary key order (see **Fig. 8**). The surrogates generated during the initial loading of a relation follow strictly the primary key order which may partially break down due to subsequent insertions and deletions; the effect of this breakdown is mitigated by allocating overflow hash slots of reduced width dynamically on an exclusive basis. As a result, the tuples of a given range of primary keys group together into one or more exclusive hash slots, and therefore relatively fewer

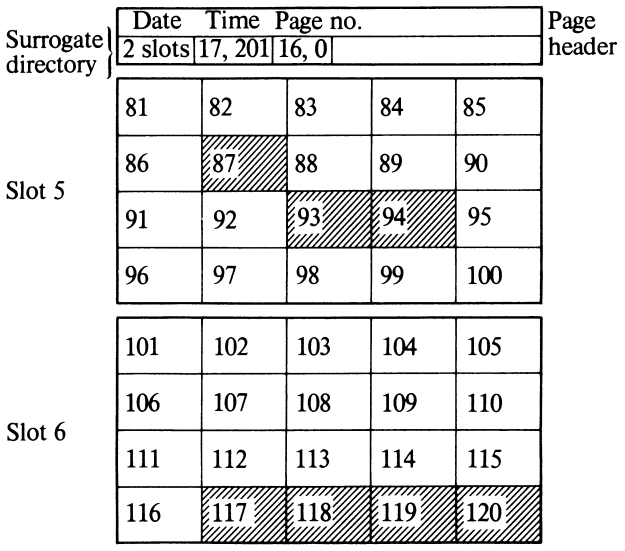
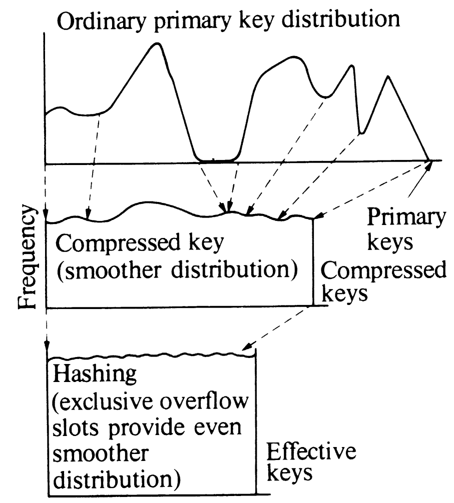


Fig. 7 Data page with W = 20 for a relation (entries showing effective keys)



Ordinary primary key distribution
Frequency
Primary keys
Compressed key (smoother distribution)
Compressed keys
Hashing (exclusive overflow slots provide even smoother distribution)
Effective keys

Fig. 8 Effect of data compression and hashing on primary key distribution

numbers of disc-accesses are required to access the tuples in primary key order than would otherwise be the case.

The shaded areas in Fig. 7 represent empty positions on a data page. As the surrogate directory shows (see also later) there are two slots, of which slot 5 has 17 tuples and its associated overflow hash slot is slot 201; slot 6 has 16 tuples but no overflow slot. Note that

$$\langle \textit{surrogate} \rangle ::= \langle \textit{rel. no} \rangle \langle \textit{effective key} \rangle;$$

entries there show only the effective keys. The size and layout of home and overflow data pages are identical except that an overflow data page would contain more hash slots if the overflow hash width is smaller. Apart from this width there is also no distinction in the surrogate directory for the two cases.

Tuples are assigned to logical pages in surrogate sequence, each page having a fixed number of surrogates belonging to one or more hash slots of the same relation; a given hash slot, but not tuple, may straddle a page boundary. The logical pages are then mapped on to data pages (physical pages) in sequence on a one to one basis. The current implementation does not permit non-contiguous allocation of hash slots to logical pages, and of logical pages to data pages, but this restriction can easily be removed if need be.

On each data page, space is reserved for all the tuples it is expected to hold. If a surrogate is not allocated or deleted, the position for the corresponding tuple remains empty; the deleted surrogates are reallocated to new tuples only after a suitable integrity check by the system. Since tuples are stored in surrogate sequence, access in that sequence is very fast. Given a primary key value, its surrogate and hence the tuple can be found from the data page directly by an estimated 1.33 disc-accesses for 30% overflows, using surrogate hashing on the primary key.

Surrogate directory

For surrogate hashing the user must specify home hash width WH, overflow hash width WO, number of home hash slots NHS and the number of overflow hash slots NOS, thus allowing WH * NHS tuples in the home data area and WO * NOS tuples in the overflow data area; default options are available for data compression and hashing techniques. Percentage of tuple density in the home hash slot during the initial loading can be controlled either by choosing a suitable value of WH * NHS or by declaration in the storage schema for IPD (Fig. 6); it can also be controlled by specific program instructions in the routine for hash code.

A *surrogate directory* (SD) containing for each hash slot the current number of tuples (C) and a pointer to its overflow slot (OS) is maintained by the system. Clearly

$$0 \leqslant C \leqslant W$$

where W = WH or W = WO as the case may be, and OS = 0 if no overflow slot is assigned. If *option* 1 is specified this directory is stored on each data page as part of a fixed length page header for the number of hash slots starting on this data page (as explained above; see also Fig. 7). Alternatively if *option* 2 is specified, the directory is stored in a compact form as part of the header in the PINDEX and contains the value (C+W+1)*OS for each slot (see **Fig. 9**). The whole of this compact directory is intended to be held in the memory during run-time for fast processing and therefore option 2 should be specified only if the expected memory available in the projected run-time environment is large enough for the purpose. For a relation of 1000 tuples the size of this compact directory is under 80 bytes and therefore option 2 would be a reasonable specification for it; on the other hand for a relation of a million tuples, option 1 would be preferable. In the examples given in this paper, option 1 is assumed unless otherwise qualified.

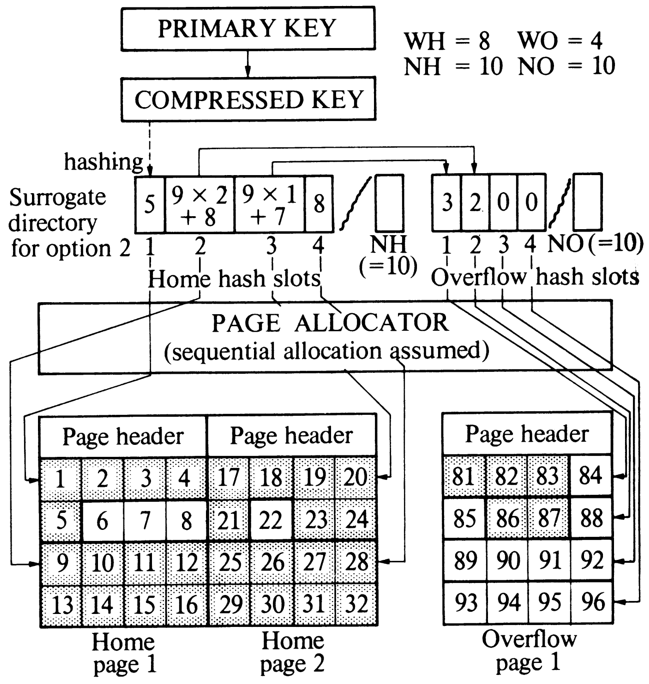


Fig. 9 Surrogate allocation

The surrogate directory is used for insertion (to get a surrogate) and deletion (to update the directory) of tuples. To insert a tuple, the compressed primary key is first hashed to find its home hash slot, the corresponding data page is then entered. If C < WH the portion of the page reserved for the tuples of this hash slot is searched (even if it has an overflow slot assigned), and the first empty tuple position found is used for the tuple. If C = WH and OS = 0, an overflow hash slot is assigned and the tuple is stored on the appropriate overflow data page in the first position of this hash slot which also determines the surrogate. If C = WH and an overflow hash slot has already been allocated, then the first available position of this overflow hash slot is used for the tuple, unless C of this overflow slot is equal to WO, in which case the process is repeated until an empty position is found (Fig. 9). For 30% overflows, an average 1·3 disc-accesses are required to find a vacancy; but if a compact directory is used, only 1 disc-access is necessary to insert a tuple, since the directory is searched in the memory before a slot with C < W is found. For deletion the surrogate of the tuple concerned is extracted from the PINDEX (described below) in order to locate the tuple on its data page (to write a deletion flag) and to find its hash slot (to update C). If C becomes zero as a result of this deletion then this hash slot is released for subsequent assignment as overflow to another hash slot (unless it is a home hash slot).

Primary key index

The primary key index (one for each relation) is basically an index of primary keys and surrogates held in the ascending or descending order of the primary key as per specification. It is used to find the surrogate of a given primary key or to access tuples in primary key sequence. The index can be designed as a B-tree (balanced tree) (Knuth, 1973, p. 473) but our standard option is what we call a *hash tree* (Deen, to be published). To construct a hash tree, the primary keys are distributed in their own sequence into hash slots of specified width using techniques similar to the one employed for surrogates. Each primary key index (PINDEX) page contains a prespecified mix of home and overflow (called local overflow) hash slots, the latter being reserved for allocation to the hash slots of this PINDEX page only. Separate overflow pages (global overflows) are also maintained. All overflow hash slots are allocated to other hash slots dynamically, no two hash slots pointing to the same overflow slot.

In the storage schema the user declares a PINDEX area and its page size. This area holds what is known as the relation information table (RIT) in which the PINDEX constitutes the major part (see also Section 9). The area available for the PINDEX proper is used for home hash slots, local overflows and global overflows. In the storage schema of Fig. 6, 30% of the available PINDEX area is reserved for global overflows, the remaining 70% being home pages with 25% of the space in each home page being reserved for local overflows; during the initial allocation only 70% of space in each home slot is to be filled. The sizes of the home and global overflow pages are the same. The user must also specify WH and WO (same for local and global overflows), and hashing and key compression techniques (default procedures exist).

Each hash slot of the PINDEX contains C (current number of entries in this slot), P (pointer to prior hash slot) and N (pointer to next hash slot), followed by C number of entries in order of their primary keys; $C \leqslant W$ where $W = WH$ or $W = WO$, depending on the hash slots. Each of these C entries consists of a triplet K*i*, S*i*, M*i* where K*i* is the *i*th primary key (held in compressed form) in this slot, S*i* its surrogate, and M*i* the current number of members in the database owned by the tuple belonging to this primary key. If this tuple is not an owner of any set in the database, then $Mi = 0$. Normally, this tuple can be deleted from the database only if $Mi = 0$, thereby protecting owner tuples from deletion in the presence of any member. An example of the content of a PINDEX hash slot is given below.

C,P,N	K1,S1,M1	K2,S2,M2	......	Kc,Sc,Mc	unused

To store a tuple, its home hash slot is derived first by applying hashing on the compressed key. If this slot is full and if it has no overflow slot, then an overflow slot is assigned—from the global overflows if a local overflow slot is not available (**Fig. 10**). Some of the entries from the original slot are moved to this newly added overflow slot in order to make the density of entries in the two slots uniform which minimises the number of disc-accesses. The new key is then placed in the correct slot position in primary key sequence irrespective of whether this position is in the original or the overflow slot. The process is reversed during deletion, removing an overflow slot if the number of entries there falls below a preset value and if there is space in the prior slot.

Using hash trees, we can find the surrogate of a given primary key by about 1·1 disc-accesses for 30% overflows; even further improvement can be achieved by reorganising the index. Since surrogates are largely in primary key order, sequential access to tuples in primary key order is fast.

Other indexes

The following three options for set and secondary key indexes are planned:

(*a*) unsorted index (kept in surrogate order except for the changes since last reorganisation; the index is intended to be reorganised periodically by a utility to provide fast access in surrogate sequence);

(*b*) B-trees (balanced trees);

(*c*) hash trees.

6. Storage usage and access speed

The data storage technique used in PRECI assumes a certain amount of storage wastage. If the distribution of primary keys is random within the range, then the storage wastage varies inversely with hash width W; and directly, but much less steeply, with the total number of tuples (NT). **Fig. 11** shows that the wastage is under 9% for 200 000 tuples with $W = 20$. As the graph flattens for high NT, the increase in percentage wastage becomes negligible for a higher number of tuples. However, in practice the distribution of keys is more often clustered than random and therefore wastage can be a little higher than that shown by the graph; on the other hand our compression and hashing techniques should offset some of the worst effects of clusters and gaps, and provide a more uniform distribution with less wastage. We

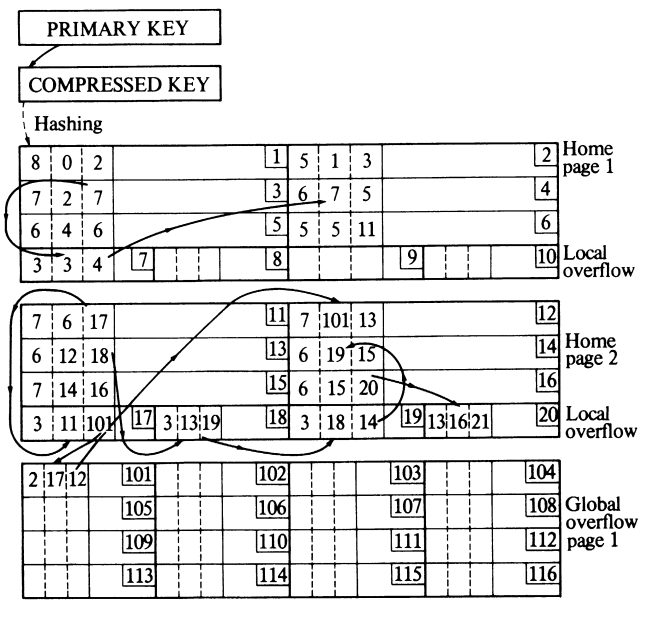


Fig. 10 PINDEX

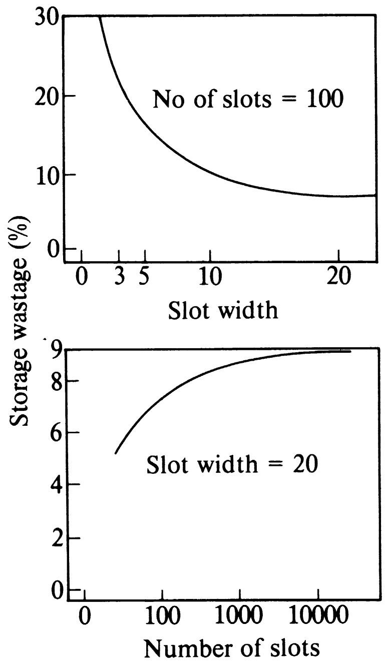


Fig. 11 Storage wastage

estimate an average storage wastage of 10–15% for PRECI in normal usage. This level of wastage appears to be acceptable to many commercial database systems, for instance IDMS (undated) wastes nearly 30% storage (the fastest direct access it provides is by location mode CALC with 1·3 disc-accesses for 30% overflows).

However, if reduction in storage wastage is a major requirement, then this can be achieved in PRECI by using the total number of home hash slots NHS = 1 which eliminates all wastage but affects the efficiency of access by primary key (see below). If the tuple population of a relation is expected to be static (retrieval oriented system), then one should make WH such that WH * *tuple size* is equal to 1 data page and NHS * WH is equal to NT (the total number of tuples in the relation), which guarantees fast access with zero wastage. However, if the population is expected to be volatile, with many insertions and deletions, then reliance has to be placed on the overflow pages; reduction in the size of the home data area decreases storage wastage but increases number of disc-accesses to the database. In PRECI the DBA can choose.

We have quoted earlier (Section 5) some access speeds for 30% overflows, meaning 30% of the tuples physically stored in the overflow data pages. This spread is, however, unrelated to whether 30% of the tuples are added later; for if 30% of the tuples are to be inserted later, then the DBA could reserve space for most of them in the home data area, reducing overflows to 10–15%. In **Table 2** some estimated access speeds are given in terms of disc-access for 10 and 30% overflows (ignoring external overheads such as those due to privacy and integrity constraints, concurrency controls etc). We assume WH = 20, WO = 10 for the data pages (*not* every home slot has an overflow slot) and 100 primary keys in the home slots of a PINDEX page. At worst, one disc-access is required per overflow hash slot of the data page, but in practice it is less, since once accessed, the overflow data page is retained in the buffer (in preference to home data pages) for further use, unless the system runs out of memory. We have, however, used the average worst case here. The figures given are in units of disc-accesses. These figures should be compared with the conventional techniques employed in most database systems which permit either random access (through hashing) or sequential access (through sequential organisation)—but unlike PRECI *not* both for the same group of records with such high access speed. As indicated earlier, 10–15% of storage wastage would normally be required to sustain this access performance; and we feel this wastage is a small price to pay—

Table 2

Access	Overflow (%) 10	30
Random (single tuples)		
By surrogate	1	1
By primary key[a]	1·11	1·33
Sequential [per group of 20 surrogates (tuples)]		
By surrogate	1	1
By primary key	1·29	1·45

[a]Using surrogate hashing on the primary key.

Table 3

Access	Overflow (%) 0	10	30
Random (via PINDEX)	2·0	2·0	2·0
Sequential [for 20 surrogates (worst case)]	1·0	2·93	6·14

particularly if judged against the figure of up to 30% wastage in commercial systems with less efficient access.

As discussed earlier, storage wastage is eliminated virtually completely if NHS = 1 (and NOS = 1) is used for the surrogate directory. This strategy leaves the speed of random and sequential access by surrogate unchanged, but affects the access by primary keys adversely depending on the level of overflow supported. The estimates given in **Table 3** assume that the PINDEX is reasonably organised with many hash slots per PINDEX page (reorganisation of PINDEX is easy). These figures would be quite typical for any system committed to 100% storage utilisation. Since most accesses and join operations are likely to be performed via surrogate or primary keys, PRECI clearly provides a basis for efficient processing giving users the flexibility to strike a balance between storage wastage and efficiency.

7. Data independence, restructuring and reorganisation

PRECI is intended to provide maximal data independence in order to facilitate restructuring of the schemas and the reorganisation of the database as needed, to incorporate new data types or to improve performance. The data independence available can be summarised as:

(1) Local schemas can be changed without requiring any changes or recompilations of the global and storage schemas.

(2) Access paths can be declared in the local schemas irrespective of whether they are actually specified in the storage schema.

(3) The storage schema can be changed without requiring any alterations or recompilation of the local and global schemas.

(4) Insertions and deletions of data items, record types or set types in the global schema do not require any changes or recompilations of the existing programs, local schemas or storage schema not referencing these items. If new additions in the global schema are not implemented in the storage schema, null values for them would be returned to the users referencing them. If any global items are deleted, the programs and local schemas using them are automatically disabled and cannot be used until recompiled.

In addition, all the data independence facilities of the CODASYL model are available as mentioned earlier.

A precompiler converts DM commands into CALL statements, and generates a tree for relational expressions (if any); this tree is subsequently used for optimisation. All data item names, record names and set names in the global schema are replaced by the compiler with unique internal identifiers which never change during the lifetime of these global objects. If a global object is deleted, its identifier will *not* normally be reallocated. The compiled versions of both local and storage schemas use these global identifiers, thus simplifying the conversion problem encountered during the binding of these two schemas which takes place during run-time, providing maximal data independence at very little extra cost.

All entries in the storage schema, with the exception of the surrogate directory, can be altered easily. For instance the reorganisation of a PINDEX need not involve any more disc-accesses than those required for the dumping of the PINDEX area to another disc. All page sizes can be altered, and new pages added. The additions or deletions of attributes, as a result of changes in the global schema, can be effected very fast by simply copying the old data pages sequentially to the new pages with the same or different page sizes; no major changes in the indexes are required. New relations and set types can also be inserted without difficulty.

The reorganisation of surrogates however is a more drastic step, as they are used as internal identifiers. In this respect PRECI is not different from other models, except that reorganisation is probably slightly easier in PRECI because of its modular structure and the exclusiveness of the range of surrogates assigned to a relation. This permits the change of the surrogates in one relation without impinging on the data pages of the others.

If surrogates are reorganised then all old surrogates must be replaced by new ones wherever they are referenced: PINDEX, secondary key and set indexes. Also tuples have to be rewritten in their data area according to the new effective key order. A utility for surrogate reorganisation is expected to be implemented at a later stage.

8. Optimisation facility

Optimisation in PRECI is expected to proceed in five stages:

(1) selection of an optimal query expression,

(2) selection of an optimal access path,

(3) deferred evaluation,

(4) indexing strategy, and

(5) buffer management.

Stages (1) and (2) are relevant to relational algebra and other very high level languages. We intend to optimise relational expressions, largely following the pioneering techniques used in PRTV (Verhofstad, 1976). Stage (2) in optimisation is the selection of an optimal access path out of those that are specified in the storage schema: SYSTEM R uses such an optimisation (Astrahan *et al.*, 1976), and we have a similar plan. We also delay the evaluation of expressions (deferred binding) until the results are actually required, thus reducing the need for intermediate storage and processing. Stage (4) provides efficient indexing facilities with options to suit differing processing requirements. Stage (5) aims at optimal buffer management during run-time with reduction in disc transfers through a flexible and efficient page swopping strategy and economic use of primary and secondary buffers (Smith, 1978).

9. Run-time system

The run-time system is being written in reentrant code to facilitate concurrent usage in the future. The system starts with the loading of the database when the database control system (DBCS) calls the initialiser to set up buffers and temporary tables and to read permanent tables from the data dictionary. Buffer space is allocated for major items:

(1) permanent tables;

(2) temporary tables;

(3) frequently used items and information;

(4) global workspace (might use secondary memory if needed);

(5) temporary indexes (partly in secondary memory);

(6) input/output area;

(7) buffers reserved for application programs (which include the subschema tables); and

(8) image buffer (mainly on secondary storage).

The system monitors the buffer usage and redistributes buffer space if necessary. Input/output is carried out in a suitable unit of buckets consisting of one or more pages. When the I/O buffer is full, the DBCS invokes a page swopping strategy.

The main permanent tables used are:

(*a*) *The master table* which contains information about other tables. (Information about temporary tables is appended to the copy of the master table in core).

(*b*) *The area table* which contains the details of areas. The copy in core holds also additional information such as the status of an area (opened, closed etc).

(*c*) *The set tables* with summary information on all set types.

(*d*) *The relation information table* (RIT)—one for each relation, which contains in the header: (1) the storage information of the relation (number of attributes, their sizes, cardinality, tuple size, keys, owner sets etc); (2) compact surrogate directory if specified; and (3) the control information on the PINDEX proper. (Home and global overflow pages of the PINDEX form the rest of the RIT.)

(*e*) *The set and secondary key indexes.*

Tables (*a*)—(*c*) are held permanently in memory, but (*d*) and (*e*) are read as necessary. Once read, the header information of RIT is usually kept in memory until this relation is CLOSEd; the pages containing detailed information are swopped more frequently.

The main temporary tables are the core table (information on disc pages held in the memory), usage table (a group of tables to hold usage statistics needed for page-swopping, integrity checking, optimisation etc.).

Temporary indexes might be created if the local access paths are not supported by the storage schema, and once created they are retained on secondary storage for the duration of the run. If changes occur to these indexes during the run, then these changes are simply appended at the end (without ordering). Work spaces are used to hold data and intermediate results for the evaluation of DM commands, particularly relational expressions. The buffer manager handles both permanent and temporary relations in a unified manner to simplify processing.

When a subschema is invoked by an application program the following entries are made in the buffer spaces allocated:

(1) subschema tables (small);

(2) header table for temporary relations (small);

(3) temporary relations (mainly on secondary devices); and

(4) system locations for currency indicators, surrogates and error status.

The currency indicators contain the current of the record type, set type etc. The programmer can also retain surrogates by a HOLDS command for using them in the same run later (see also Section 4.) Any error detected during the execution of a DM command is flagged by the database control system with its error code in the error status indicator which the programmer can check.

The areas associated with each relation are opened and closed by an explicit OPEN/CLOSE *relation name* command issued from the application program. When a CLOSE command is encountered, the areas are checked for integrity and all the relevant buffers (after images) are written out as necessary. In the current version only an elementary backup/recovery facility—with after images, checkpoints and dumping—is provided. During an update all after images are written out; the user may request for checkpoints during run-time, but database archiving is carried out only by special runs. A trace facility can be invoked for debugging.

During run-time the deleted tuples are flagged, but the surrogates released are not immediately reallocated. This ensures that other users (in a concurrent environment) requesting a deleted tuple get an appropriate error message, rather than a newly inserted wrong tuple in the same position.

No distinction is made by the run-time system between the various interfacing models except at the DM command level. If an algebraic command is encountered, the relevant DM tree is read and the optimisation facilities (1)–(3) are invoked (Section 8). For a CODASYL FIND command, optimisation (2) is used. Format conversion between the subschema and storage schema is carried out in both directions as and when necessary.