AN IMPLEMENTATION OF IMPURE SURROGATES

S. M. Deen

Department of Computing Science
University of Aberdeen
Aberdeen, Scotland

ABSTRACT

Surrogates or internal identifiers can be made to facilitate both fast access and storage independence if they are implemented properly. Such an implementation is discussed here; it permits the tuples of a relation to be accessed very fast by primary key for both random and sequential search without retarding the performance of secondary keys. It employs a key compression and a hashing algorithm and attempts to place tuples on data pages in the primary key sequence. Subsequent updates are absorbed by a dynamic allocation of overflows. An indexing technique called a hash tree holds surrogates in primary key order, and facilitates fast sequential access. The access speed remains high even at a 90% load factor, without being significantly affected by storage reorganisation resulting from the addition of new attributes, deletion of old attributes or change of data page sizes.

These techniques have been implemented in the PRECI database system at Aberdeen.

1. INTRODUCTION

The basic concept of surrogate as an unique internal identifier is not new, although this particular term is. It has been used in various forms and names in a number of database products - in the Codasyl model it is called a database key, in Adabas an internal sequence number, and in SYSTEM R a tuple identifier. The concept has been fully defined by Hall et al [1] who also advocated the term surrogate. Codd recognised it in [2].

Hall et al have idealised a surrogate as that identifier of an entity which never changes. For instance the surrogate must not change if an employee changes his/her name. If a course is now given by a different teacher who has changed both the title and the syllabus of the course (quite a common occurrence in a university) it somehow still remains the same course, and hence must have the same surrogate. We have however deviated from this idealised concept in our implementation, and hence the notion of impure surrogates, as will be explained below.

In this paper we assume a primary key (pkey for short) not only to act as the unique identifier of tuples, but also to be that key which is used most frequently for random and sequential access to the tuples. A surrogate is the permanent internal identifier of a tuple; we can generate it independent of the relevant pkey, and if this is done, it has to be linked to the pkey value by something like a Balance tree (B-tree) [3]. However, such a technique is ineffecient as it can take several disc accesses to find a surrogate from a large B-tree, and at least another disc access to retrieve the tuple. We therefore use an alternative strategy and generate surrogates from the pkey values for faster access. This however means that if a pkey value changes, the corresponding surrogate must also change. Thus we deviate from the original pure concept of permanent surrogates. The disadvantage of this impure concept is trivial compared to its advantages. At worst we have to delete and reinsert a tuple if its primary key value changes; but it provides fast random and sequential access by pkey, and also allows a flexible storage structure permitting additions/deletions of attributes and change of page sizes without any significant loss of retrieval efficiency.

It is possible to store tuples in strict primary key sequence by using a combination of hashing and page-splitting techniques [4]. The pkey value is hashed to determine the data page where the tuples are held in that key sequence; in the event of an overflow the content of the page is split redistributing the tuples over two pages - the original page and an overflow page - again in the same key order. Use of hashing avoids the need for large indexes, and the technique permits very fast access by pkey both randomly and sequentially. However accesses by secondary keys become much slower, due to the need of a large index and/or frequent reorganisations. The secondary keys can be supported there by an index either for (i) (secondary key, primary key) pair, or for (ii) (secondary key, storage location) pair. Option (i) tends to make the index large as the size of the pkey can be large, and this in turn slows down the processing. The replacement of a primary key by a

compressed primary key does not work since the data compression is not a reversible process, that is, we can not derive a primary key value from its compressed form without additional information. Option (ii) needs frequent reorganisation of the index, at least once after every splitting, and hence unacceptable particularly in a relation where a large number of secondary keys are supported. What we need is a (secondary key, surrogate) index where a very fast access by surrogates is guaranteed - ideally as fast as the access by storage locations, but at the same time preserving storage independence mentioned earlier. This is precisely what we allow for, but secondary key implementations is not covered in this paper. In our model,

We can not get a storage location directly from any key value (primary or secondary) without going through the surrogates. The following surrogate facilities are needed:

(i) Surrogate generation

When a tuple is inserted, a surrogate must be generated and the surrogate directory updated.

(ii) Surrogate release

When a tuple is deleted, the surrogate directory must be updated, releasing the surrogate for possible re-use.

(iii) Surrogate access

Given a pkey value, the system should be able to find the surrogate.

(iv) Storage access

Given a surrogate, it should be possible to find the stored tuple.

An efficient and flexible implementation of the surrogate facilities demands the following:

(i) Fast direct and sequential access to tuples by pkey.

(ii) Fast direct and sequential access to tuples by surrogates. Fast direct access by surrogate is essential if we are to provide fast access by the pkey and secondary keys. The fast sequential access by surrogate can be used to advantage by an intelligent DBMS for fast access to a set of tuples (not necessarily contiguous) yielding the same secondary key value. It can also be used to support an efficient system defined order, as in the Codasyl set order clause.

(iii) Independence of surrogates from storage locations.

(iv) Low storage wastage.

Some of these requirements might dictate trade offs; we claim that our technique satisfies the requirements to a high degree as discussed in this paper. The plan of the paper is as follows. In section 2, we describe our surrogate generation and tuple storage technique, and indicate the flexibility they provide in storage reorganisation. The next section explains the surrogate directory and the primary key index, the latter holding surrogates in pkey order. The storage usage and access efficiency are considered in section 4 where a brief comparison with SYSTEM R and the Codasyl model is also attempted. Section 5 contains a conclusion.

2. SURROGATE GENERATION AND TUPLE PLACEMENT

In this model, a surrogate is constructed as a concatenation of an internal relation number (irn) and an effective key value (ekey value) as:

surrogate ::= <irn> <ekey value>

where ekey is generated from pkey - using a hashing and a key compression algorithm, supported by an overflow mechanism. The resultant surrogates are then allocated to physical pages, referred to as data pages. These techniques are applied separately to each relation, with different parameters, and hence in their description below we shall assume only one relation unless indicated otherwise.

The ekey value gives the relative position of a tuple in a stored relation in insertion time sequence, except where a surrogate is re-allocated after the deletion of the original tuple. Therefore by sorting the tuples to be inserted in pkey order at the first loading of the relation, we make the surrogate and pkey order coincide. However the subsequent insertions of tuples in the same relation are potentially unordered, and therefore there is a need for an index (referred to as PINDEX) to yield surrogates (and hence tuples) in pkey order as discussed in section 3.

2.1 Hashing Technique

The hashing technique we use is called a division hashing, which when applied to a pkey value gives a quotient, referred to as surrogate home (hash) slot or SHS for short. The word 'home' distinguishes it from the surrogate overflow (hash) slot or SOS discussed later. The expected average number of tuples in SHS is fixed for a given relation, and is called surrogate home (hash) width (SHW). To explain the hashing

technique, we consider an example.

Assume we have a 4-digit pkey for a given relation with 10,000 possible pkey values (called pkey range K) ranging from 0000 to 9999. Of these 10000 potential values, suppose we expect only N values (i.e. N tuples) ever to be loaded in the database for this relation. We divide N by H number of surrogate home slots to get SHW. Say N=1000, and H=50. We define a divisor D as

$$D = \frac{K}{H} = \frac{10000}{50} = 200$$

$$SHW = \frac{N}{H} = \frac{1000}{50} = 20$$

The ekey value corresponding to a pkey value p is then

$$\text{ekey value} = \frac{p}{D} * SHW + C$$

where we take only the integral part of the result p/D and $0 \leq C \leq SHW$, C being the current number of tuples (including this tuple) in this hash slot. In other words C is the relative position of this tuple in this hash slot in insertion time sequence. [We shall ignore here the situation where a surrogate is re-allocated after the deletion of the original tuple - it is discussed later].

Assume that we are storing the relations specified above for the first time with the following pkey values:

2, 3, 50, 103,189, 204, 251, 278

Then SHS = 1 for pkey values 2, 3, 50, 103 and 189
with C = 5 and the corresponding ekey values 1, 2, 3, 4 and 5 respectively.

SHS = 2 for pkey values 204, 251, 278
with C = 3 and the corresponding ekey values 21, 22 and 23 respectively.

In a subsequent run if we have pkey = 23 and 300, then the corresponding ekey value will be 6 (C=6, SHS=1) and 24 (C=4, SHS=2) respectively. Note that in the first slot (i.e. SHS = 1) the pkey order is not maintained, but in the second slot it is. We do not change surrogates for the preservation of the pkey order, since such a change would involve a major reorganisation of all indexes and storage positions.

One of the problems encountered in a division hashing algorithm is the high collision probability, that is, the chance that too many pkey values might yield the same SHS. We apply two remedies to control the situation.

Key compression
Overflow slots

as explained in the next subsections.

2.2 Key Compression Technique

In key compression, the pkey values are compressed to provide a more uniform distribution, while retaining the original key order. For instance if an organisation has employee numbers in three ranges 11 to 90, 201 to 300, 501 to 900, then these ranges can be replaced by the following three ranges: 1 to 80, 81 to 180 and 181 to 581 - the final overall compressed key range being 1 to 581 instead of the original overall key range 11 to 900. The mapping between the original and compressed key range would look like:

Original key values	Compressed key values
13	3
15	5
18	8
⋮	⋮
205	85
214	94
225	105
⋮	⋮
510	190
517	197
521	201
⋮	⋮

The compressed key range is clearly more uniformly distributed. The divisor D given earlier should now be obtained by dividing the compressed key range (rather than the original pkey range) by H. The sucess of the compression algorithm depends on the prior knowledge of the pkey value distribution, and on the effectiveness of the algorithm itself. The generated code has to be reasonably small so that it can be held in the memory during run-time. A generalised key-compression technique to deal with non-uniform distributions is given in [11], but not discussed here.

2.3 Surrogate Overflow

If the pkey value distribution is assumed to be non-uniform or Poissonian, then in the absence of any key compression, we would expect some SHSs to have too few and some too many values. A good compression technique will reduce the under and overflows, but is unlikely to eliminate them completely. Underflows lead to storage wastage and overflows access inefficiency, and one must strike a balance. For overflows, we provide HO number of surrogate overflow (hash) slots (SOS) with a fixed (for a given relation) surrogate overflow (hash) width (SOW).

An overflow slot is allocated dynamically, to an over-flowing slot which can be a home slot or another overflow slot. The allocation is exclusive that is the same overflow slot can not

be shared by two overflowing slots, nor can an overflowing slot have two overflow slots directly linked to it. If overflow slot B of an overflowing slot A overflows then the next free overflow slot is allocated to overflow slot B (which is now overflowing) but not to the original overflowing slot. Note that an overflow slot is allocated only when a slot actually overflows, and not before. Once allocated an overflow slot can not be removed until it becomes empty (that is free) due to, say, subsequent deletions of tuples; a free overflow slot can be allocated to any needy overflowing slot. The ekey value of the first tuple in an overflow slot SOS_i is

$$H * SHW + (SOS_i - 1) * SOW + 1$$

irrespective of the overflowing slot to which this overflow slot SOS_i is allocated. A second tuple falling into this slot will get the next ekey value, and so on. The last tuple of this slot will be the (SOW)th tuple in insertion time order, and will have the ekey value

$$H * SHW + SOS_i * SOW$$

We define the percentage of surrogate overflow as 100 * HO * SOW/(H*SHW).

If SOW = 1, then the storage wastage is minimal, but the overhead is high. A size SOW = 1/3 SHW or SOW = 1/2 SHW seems more reasonable for random distribution. In our examples we shall mostly use SOW = 1/2 SHW for convenience.

2.4 Tuple Placements

Hash slots are allocated to physical data pages as per an algorithm. The simplest method is to allocate them in their own order: SHS_1, SHS_2.... SHS_H, SOS_1, SOS_2....SOS_{HO} - which maintains the effective key (and hence surrogate) sequence. However the allocation can be non-contiguous (but not considered in this paper) within this sequence if the data pages are shared with the hash slots of other relations, each hash slot having an exclusive portion of storage space (called storage slot) for its tuples. A hash slot may straddle a page boundary. If tuples of a relation are of fixed length, as customary in most relational implementations, then given a surrogate we can directly determine the physical location of the corresponding tuple from knowing the page size. This requires no indexes or disc access overheads, hence the tuple can be retrieved by a single disc access. We assume all data pages of a relation, irrespective of whether they hold home or overflow slots, are of the same size.

The above prescription of tuple placement with contiguous allocation is displayed in figure 1 where SHS = 11 and 12 of relation 05 occupy page slots 11 and 12 (top and bottom half of page 6) respectively. We assume the following parameters for the relation 05.

pkey range K	= 10000
maximum number of expected tuples N	= 1000
maximum number of SHS	= 50
SHW	= 20

With 40 tuples per page, we need 25 data pages.

Let us suppose that only 10 tuples with ekey values 201-210 are stored on that page slot at first loading in pkey order, the remaining positions being empty. The next tuple into that page slot will get the surrogate 05211 unless an earlier surrogate is available due to deletion. For instance if the tuple with surrogate 05206 is deleted and if surrogate is free (the surrogate of a deleted tuple is not necessarily immediately freed for reallocation due to integrity reasons), then our new tuple will get this surrogate rather than 05211. In either case the stored position of this new tuple might not be in the correct pkey sequence. The placement of an overflow hash slot is similiar except that there will be more overflow slots per data page due to their smaller slot widths. A more detailed description of insertion/deletion/retrieval operation is given in the next section.

If the tuples of the same relation vary in length, then our overflow technique described above will not work. For such cases, we shall probably divide an overflow page into fixed length sections, and allocate dynamically one section to each needy overflowing page - exactly in the same way as for the surrogate overflow slots. Either the excess length of the tuples, or some of the tuples can be transferred to these overflow sections. The processing for these tuples will naturally be slower. We have not implemented this technique but might investigate it in a later paper; however in the rest of this paper we shall assume all the tuples of a relation to have the same length, unless otherwise indicated.

2.5 Data Page Reorganisation

Since tuples are stored on surrogate sequence, they can be easily copied in the same sequence on to other pages of smaller or larger size without any significant copying overhead. The surrogate directory and PINDEX (both described later) are not affected. Therefore if a new attribute is added or an old one deleted, the tuples are simply copied out on to new data pages with the new tuple length. This avoids the need for any pointers and retains largely the original access efficiency (see section 4).

3. SURROGATE DIRECTORY AND PKEY INDEX

We need one surrogate directory and a pkey index (PINDEX) for each relation, and both are used

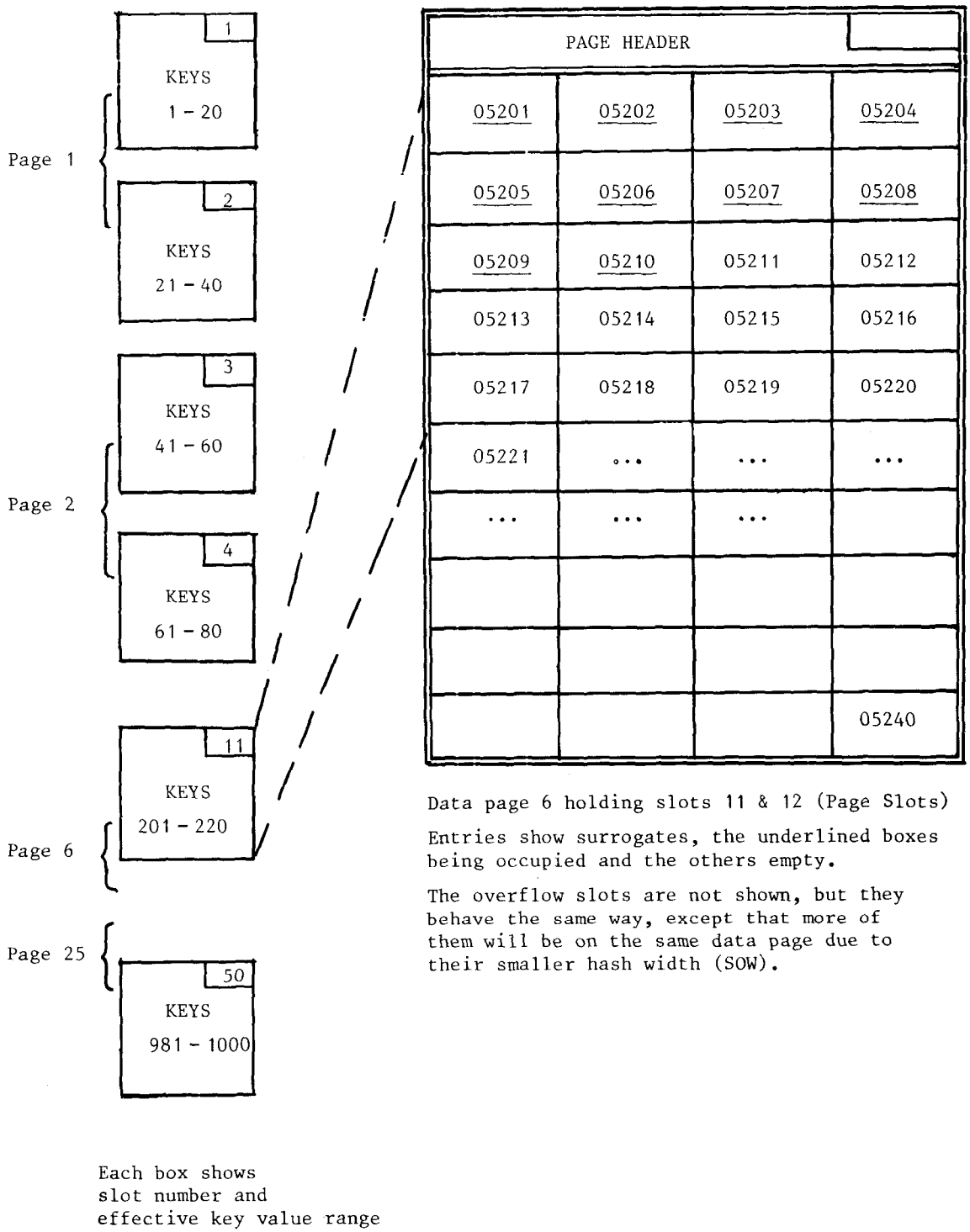


FIGURE 1: TUPLE PLACEMENT

during insertion, deletion and retrieval of tuples.

3.1 Surrogate Directory

A surrogate directory is accessed during the run-time for three functions

(i) to allocate a surrogate
(ii) to release a surrogate
(iii) to access a tuple directly by a pkey value.

The directory basically consists of one entry for each hash slot (both home and overflow), each entry containing the current number (C) of tuples in that slot, and its overflow slot number (SOS) if allocated. We provide two options for the directory: option 1 for the compact surrogate directory (CSD) and option 2 for the disperse surrogate directory (DSD). The CSD of a relation is small and can be held in the memory during the processing of the relation, whereas the disperse directory is dispersed on the data pages, each data page holding the directory information for the hash slots of this page. Our standard option is CSD and is described first.

In CSD, the entry for each hash slot i has the following value V

$$V = (W+1) * SOS + C$$

where W = SHW or SOW depending on whether i is a home or an overflow slot, SOS is the overflow slot allocated to i and C is the current number of tuples in slot i. Clearly $0 \leq C \leq W$. If no overflow slot has been allocated then SOS = 0, and hence V = C. The entries for a CSD are shown below with SHW = 10 and SOW = 5.

	3	10	21	8	31
slot numbers	1	2	3	4	5

Home slots

	2	29	0	1
slot numbers	1	2	3	4

Overflow slots

In home slot 2 we have V = 10 = 11 x 0 +10 = C, and hence this slot is full, but no overflow slot is allocated to it. However in home slot 5, V = 31 = 11 x 2 + 9, and hence its overflow slot number is 2 and C = 9. This is possible - it means that a tuple is deleted from the home slot after the allocation of an overflow slot. This deletion does not affect the overflow slot, but the next insertion will be in the home slot since C = 9 < SHW. The overflow slot 2 is full and has got an overflow slot (slot number 4) since its V = 29 = 6 x 4 + 5. In home slot 3 V = 21 = 11 x 1 + 10 indicating that overflow slot 1 has been allocated to it.

To insert a tuple, the pkey value is compressed and hashed to find the surrogate home slot. If C < SHW there, then the relevant page slot is sequentially scanned for an empty position. The first empty position found provides the surrogate and storage space for the tuple. If C = SHW, and SOS = 0, the first free surrogate overflow hash slot (from a bit map maintained for the purpose in the directory header) is assigned. The tuple concerned then becomes the first inhabitant of this overflow hash slot. If however an SOS has already been assigned and its C < SOW, then the relevant page slot is scanned for an empty position as above: if C = SOW, then another overflow slot is assigned to this overflow slot (but not to its home slot), and the process is repeated. Note that we need only 1 disc access for this search irrespective of overflows since the directory is searched in the memory. To retrieve a tuple directly by pkey, the surrogate home slot is found as above and then the relevant page slot scanned for a match of the pkey value. If it is not found, then the page slot of the relevant overflow hash slot is searched. The process is repeated for all the relevant overflows until the tuple is found or end of overflows reached. For 30% surrogate overflows, we therefore need 1.33 disc accesses for retrieval. For deletion, the surrogate is obtained from the PINDEX since the PINDEX has to be entered for all insertions and deletions anyway; it takes only about 1 disc access to get a surrogate from a well-organised PINDEX as described later. Using the surrogate, the tuple position is retrieved from storage by a further disc access. A deletion marker is then written and the surrogate directory updated decreasing C by 1. If C of an overflow slot becomes 0, then the slot is released for subsequent reallocation after a suitable integrity check.

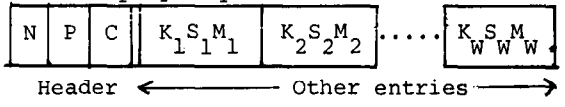
The Compact Surrogate Directory described above is small - taking for instance about 80 bytes for a relation of 1000 tuples - and hence can be kept in the memory during its processing as assumed above. For very large relations, say a million tuples and 100000 slots the CSD can be too large for the memory, and in that case our second option, the DSD applies. There C and SOS of the surrogate hash slots (both home and overflow) are maintained on the data pages for each hash slot starting (not necessarily ending) on this data page. This is of course less efficient. For instance, to find an empty position for the insertion of a tuple, we will first have to access the data page of the home slot to examine the directory. If C = SHW, we must go to the data page holding its overflow hash slot (if any) and so on. Thus for 30% surrogate overflows we would need 1.33 disc accesses here, as against 1 disc access in the case of CSD.

3.2 Pkey Index

Since some of the tuples are not stored in pkey order, we need an index referred to as PINDEX,

to access the tuples in pkey sequence. This index can be a B-tree, but we describe here an alternative technique called a hash tree which provides faster access than by a B-tree of depth higher than two (see Section 4.2).

In a hash tree, the pkey value is compressed and hashed to find a slot where this pkey value and its surrogate are held. The algorithms for PINDEX are generally assumed to be different from those for the surrogates, except that in both the compression algorithm must retain the original key order, and that the hashing technique must be division. As in the case of surrogates, the quotient of the division hashing gives the PINDEX home (hash) slot (PHS) with a fixed PINDEX home (hash) width (PHW). PINDEX overflow (hash) slots (POS) with a fixed PINDEX overflow (hash) width (POW) are also provided. Each slot (home or overflow) contains a header followed by a set of pkey entries in pkey sequence as shown below:



where: N(P) is the next (prior) PINDEX slot (home or overflow) in pkey sequence, and C is the current number of pkey values in this slot, with $0 \leq C \leq W$ where W = PHW or POW as the case may be.

$(K_i S_i)$ are the compressed key value and the surrogate of ith pkey value in this slot. The compressed pkey value is used to reduce the index size.

M_i is the number of other tuples (members in the Codasyl sense). This tuple can not normally be deleted unless $M_i = 0$, and hence M_i permits an integrity check; but M_i can be dropped from the index if desired.

The entries $(K_i S_i M_i)$ are held in strict pkey order, with new insertions being placed in the correct ordinal positions. As in the case of the surrogate directory, the overflow slots are allocated dynamically and exclusively to needy overflowing slots (home or overflow). However when an overflow slot is allocated the entries are spread evenly in pkey sequence among the overflow slot. For instance, suppose PHW = 8, POW = 4, and say a PINDEX home slot has the following 8 pkey values

21	23	42	56	58	60	62	68

If we insert a key value 30, we must allocate an overflow slot with the following result

21	23	30	42	56	58	E	E

Original Home Slot

60	62	68	E

Overflow Slot
E for empty position

This minimises the need for slot splitting for Poissonian or non-uniform insertions. The process is reversed for deletions. The evenness of the key value distribution between the overflowing and overflow slots is maintained irrespective of whether the overflowing slot is a home or an overflow slot.

In a hash tree we provide two types of overflows: local and global, both having the same POW, but stored on different types of PINDEX pages. PINDEX home slots along with the local overflow slots are stored on what are called PINDEX home pages, and global overflow slots on PINDEX global overflow pages (Figure 2). Except this distinction in contents, there is no other difference between these two types of pages, for instance, both have the same page size. On each home page, a certain proportion of space, say 25%, is reserved for local overflow slots to be allocated only to the overflowing (home or local overflow) slots of this page. A global overflow slot is allocated only when all the relevant local overflow slots are occupied. (Both the local and global overflow slots are allocated on dynamic and exclusive basis with only one overflow slot being directly linked to an overflowing slot). The existence of local overflows on the same PINDEX page reduces disc accesses.

The PINDEX can be reorganised quite easily without affecting the rest of the database. Such reorganisation should be done periodically with different hash widths and overflow distribution, partly to reduce global overflows. As the index is relatively small the load factor can be kept at around 70 to 80% without wasting too much storage space. This should keep the global overflow low and permit the retrieval of a surrogate for a pkey value by about a single disc access. The PINDEX is updated during insertions and deletions of tuples and is used for sequential processing of tuples by pkey. Sometimes it is also employed for random access by pkey (see later).

In a highly clustered key distribution where wide gaps are known to exist, the PINDEX home slots are numbered in a non-contiguous, but ascending order. For instance if we do not expect any key value for slots 6 to 36, then these slots will not be created, instead slot 37 will follow slot 5. If any key later yield an intervening missing slot, it will be homed in slot 37. This technique is called slot collapsing, and requires each PINDEX home slot to contain its slot number in the header which in that case will have an extra field. This collapsing technique can also be used for surrogates, but a more compact version (since the CSD is kept in the memory) is proposed in ref [11].

4. STORAGE USAGE AND ACCESS EFFICIENCY

The storage usage and access efficiency are closely related, one can be gained at the expense of the other. For instance if the total number of expected tuples N in surrogate hashing is assumed to be higher than it is, then there will be an increase in underflows and a decrease in overflows resulting in higher efficiency in direct access by pkey. We wish to examine here the storage wastage our technique incurs and the access efficiency it provides - along with a brief comparison with two other techniques.

4.1 Wastage Calculation

The storage space wasted on data pages depends mainly on the success of the compression algorithm and the hashing technique. The compression algorithm is an unknown entity, but the impact of hashing can be easily ascertained if we assume a random distribution of pkey values, as done below.

Let us assume that N is the maximum number of pkey values out of the pkey range K to be distributed over a maximum number of surrogate home hash slots H where H = K/D >> 1, with surrogate home hash width SHW = N/H. The probability p[r] that there are exactly r tuples in a slot is:

$$p[r] = \left({}^{D}C_{r} * {}^{K-D}C_{N-r}\right) \Big/ {}^{K}C_{N}$$

$$\text{with } \sum_{r=o}^{D} p[r] = 1 \qquad (1)$$

The number of slots getting exactly r tuples = H*p[r]. The space wasted (unit is the number of tuples) if there are only r tuples in a slot is (SHW-r). Therefore the wastage by the slots that have exactly r tuples is H*p[r]*(SHW-r).

Hence the total space wasted

$$TW = \sum_{r=o}^{SHW-1} H*p[r]*(SHW-r) \qquad (2)$$

The percentage wastage in the home slots

$$= TW/N = TW/(H*SHW)$$

The overall percentage wastage

$$= \frac{TW + \text{wastage in the overflow slots}}{H*SHW + HO*SOW} \qquad (3)$$

$$\simeq TW/N \qquad (4)$$

Since the wastage in the overflow slots is expected to be less than that in the home slots, the eqn (4) is an upper limit.

Using expressions (1) and (4) we have calculated the percentage wastage for a number of cases. The two plots presented in figure 3 are:

(a) Percentage wastage against the value of slot width SHW for K = 10000, N = 1000

(b) Percentage wastage against N for K = 10000 and SHW = 20

Plot (a) gives the wastage of 12, 9.5 and 8.5% for SHW = 10, 15 and 20 respectively, the wastage reducing further but more slowly for higher values of SHW. In plot (b) the maximum wastage is 8.5% for K = 10000, irrespective of N. Similar calculations with larger values of K (K = 100000 and 1000000) did not affect plot (a) or the value of the maximum wastage shown by plot (b) in any significant way. Thus a wastage of 8.5% appears reasonable. The dynamic allocation of overflows and an effective data compression technique should improve this situation. Note that if SHW = N (hence H = 1), the wastage is 0%.

4.2 Access Efficiency

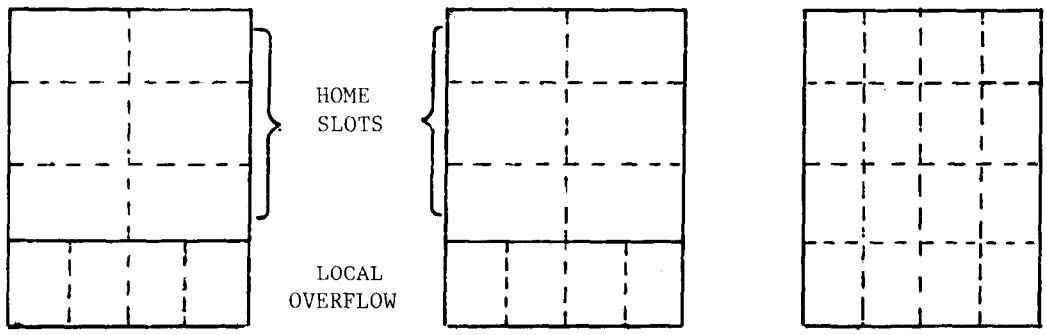
We wish to present here some theoretical estimates of access efficiency for a given percentage of surrogate overflows in our model. The figures given will be evaluated for 0, 10 and 30 percent surrogate overflows, with SHW = 20, SOW= 10 (not relevant for 0% overflow), PHW = 10 and POW = 5. It is assumed that PINDEX is reasonably organised with 10 PINDEX home hash slots and 5 PINDEX overflow slots per PINDEX home page, thus each such home page having up to 100 surrogates in the home slots, and 25 surrogates in the local overflow slots - with, say, an average population of 100 pkey values per page. We also suppose that the global overflow area is 10% of the PINDEX home pages. Note that since PINDEX can be reorganised, its overflows (local and global) are expected to be smaller than the surrogate overflows. We shall consider below random and sequential accesses by surrogates and pkey. The unit used to measure access speed is a disc access.

The speed for random access by surrogate is 1 disc access irrespective of surrogate overflows, and that by pkey is 1.11 for 10% and 1.33 for 30% surrogate overflows as pointed out earlier (see also below). Sequential access by surrogates is fast with 1 access per data page (assuming hash slots are placed in their own sequence). Sequential access by pkey has two components:

(i) PINDEX access time T[P] and

(ii) data page access time T[D]

If the PINDEX is reasonably organised, as we have assumed it is, the need to access PINDEX global overflow should be neglible in the case of 10% surrogate overflows. Therefore for 0 to 10% surrogate overflows we need 1 disc access, and for 30% surrogate overflows 1.10 disc accesses (since global overflow is 10%) to retrieve 100 surrogates for 100 pkey values (average home page population) from PINDEX. Hence time T[P] for 20 surrogates is 0.20 and 0.22 for up to 10% and for 30% surrogate overflows respectively. (Note



PINDEX Home Page
6 Home and 4 Local Overflow
Slots per page

PINDEX Global
Overflow Page
16 Overflow Slots

POW is same for both overflows and assumed to be 1/2 PHW here.

FIGURE 2: PINDEX PAGES

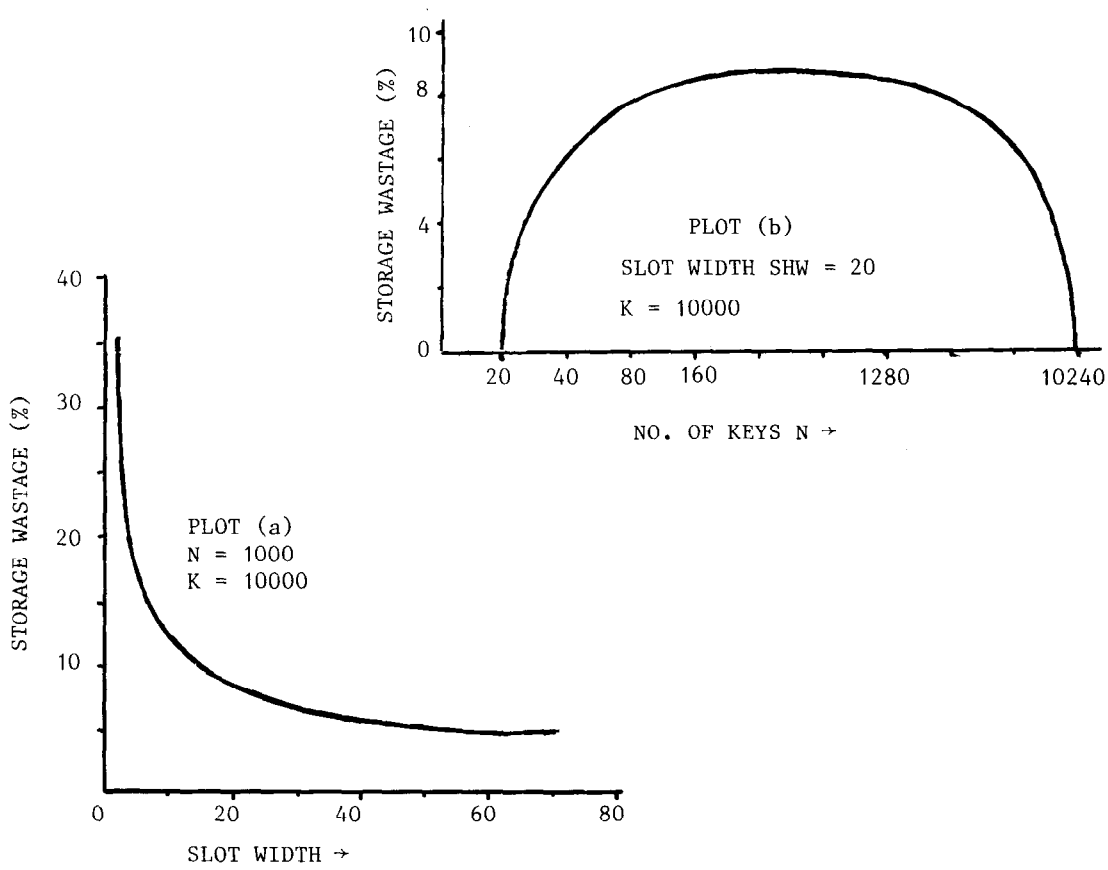


FIGURE 3: STORAGE WASTAGE

that if we wish to get the surrogate of a single tuple from the PINDEX, then according to these assumptions, we need 1.0 disc access for 10% and 1.1 disc access for 30% surrogate overflows; in contrast, irrespective of surrogate overflows, a two-level B-tree will require 1 disc access and a three-level B-tree 2 disc accesses, if the root is held in the memory.)

For x percent surrogate overflows, we have 2 * x overflow slots in every 100 surrogate home slots, since SHW = 2 * SOW. To access the tuples of 100 surrogate home slots and 2*x surrogate overflow slots - holding 20*(100+x) surrogates altogether - we need 100 + 2 * x disc accesses. Therefore time T[D] to access 20 surrogates is (100+2*x)/(100+x). The total access time is then:

$$T[P] + T[D] = T[P] + (100+2*x)/(100+x)$$

The values for different percentages are shown in table 1. The figures given there should remain valid up to a 90% load factor since the anticipated storage wastage is only 8.5% as discussed in the last subsection.

TABLE 1

Unit is a disc access

	Overflow on data pages		
	0%	10%	30%
Random access			
by surrogates	1	1	1
by pkey	1	1.11	1.33
Sequential access			
by surrogates (per data page)	1	1	1
by pkey (per 20 tuples)	1.20	1.29	1.45

Random access (but not sequential access) by pkey will be slower if hash slots cut across page boundaries. It is assumed that at least initially slot width will be so chosen that an exact multiple of hash slots is held on a page. However if tuple size expand/contract by a fraction of the original size - rather than being doubled/halved - due to say additions/deletions of attributes, then the page alignment will be lost, slowing down random access by pkey. However, at worst - whether due to excessive surrogate overflows or loss of page alignment - we can always access a tuple via PINDEX with 1.1 disc accesses (as estimated earlier for 30% overflows) to get the surrogate and 1 disc access to get the tuple by surrogate. In that case random access by pkey will require 2.1 disc accesses, perhaps reducing to 2 disc accesses if the PINDEX is reorganised very efficiently. In our present implementation in the PRECI database system [9, 10] PINDEX is used for random access by pkey if surrogate hash slots straddle page boundaries. It is also possible to reserve initially more space in the surrogate home slots (by just having a larger N or larger tuple length), which will reduce the size of surrogate overflows and the need to cross page boundaries.

The storage wastage can be eliminated virtually completely if we use SHW = N and H = 1 with no surrogate overflow slots; in other words a single surrogate slot for the whole relation. In this case, the speed of access by surrogates will not be affected, but those by pkey will be. For a random retrieval by pkey we will need 2.1 disc accesses, as explained above. Sequential access by pkey will be expensive since the stored tuples will not generally be in pkey sequence.

4.3 Comparison With Other Models

We shall briefly compare our technique with those employed in SYSTEM R and Codasyl implementations.

In SYSTEM R [5,6,7] each tuple has a tuple identifier (TID) made up of page number and page offset, and hence storage can not be easily reorganised outside the page. The user can suggest a TID during insertion, but if the suggested page is not available, the tuple will be stored on an adjacent page. The TIDs are linked to key values by a B-tree. SYSTEM R permits variable length tuples and dynamic addition of new attributes, both of which can lead to overflows. If the original page is full, an overflowing tuple is moved to another page with a tag in the original location giving the address of the new location. In that case two disc accesses are necessary to access the tuple by its TID.

Comparison with SYSTEM R depends on a number of assumptions, such as page size, key length, depth of the B-tree etc. A SYSTEM R page is 4096 bytes which can be assumed to hold 200 (key, TID) entries. If the cardinality of a relation is under 40K, then a two-level B-tree will suffice, with a three-level B tree for above 40K. The root of the tree may be assumed to be in the memory, thus requiring 1 disc access for a two-level and 2 disc accesses for a three-level tree. In addition, we need 1 to 2 disc accesses to retrieve a tuple by the TID obtained from the B-tree. SYSTEM R does not support primary key as such, but we can assume a unique key with clustering index as the closest equivalent of our primary key. We shall consider two cases, one with cardinality 30000 and the other with 50000. We shall assume the compact surrogate directory, which can take up to 2 pages for 100000 tuples, to be in the memory as well.

Random Access by key or equivalent

Columns (i) of Table 2 refers to fixed length tuple, retrieved in a single disc access by TID. Calculations for our model follow those made for Table 1.

Over-flow	No. of tuples	SYSTEM R		Our Model (PRECI)	
		(i)	(ii)	(i)	(ii)
0%	30K	2.0	3.0	1.0	2.1
30%	39K	2.0	3.0	1.3	2.1
45%	45K	3.0	4.0	1.5	2.1
0%	50K	3.0	4.0	1.0	2.1
30%	65K	3.0	4.0	1.3	2.1
45%	75K	3.0	4.0	1.5	2.1

Table 2

Access in our case will be faster, if space is reserved initially in surrogate home slots, which will reduce the overflow.

Addition of new attributes

SYSTEM R will require 2 accesses by TID assuming the original pages as full. This will increase the SYSTEM R figures to those under column (ii). Our figures will change to a fixed 2.1 access (column ii), as explained earlier, irrespective of overflows and cardinalities. If the original pages are so populated as to allow some later growth without overflowing (in our case it is equivalent to the use of larger tuple length initially) then figures of Table 2 could remain unchanged for both the models.

Sequential access

Intuitively sequential access by pkey is faster in our case since no tuple is spread over two pages and since tuples are clustered in surrogate home and overflow slots. We also do not have the overhead of SYSTEM R Prefix with each tuple. Any detailed comparison depends on too many assumptions, and hence not attempted.

In the above comparisons, tuples are assumed to be fixed length. As we have not investigated the problem of variable length tuples in our model, we can not compare this case, but intuitively its affect in our model would be like that of the overflows in Table 2 and in SYSTEM R it would increase the access by TID to more than 1 disc access. There are also other forms of accesses which we have not considered; SYSTEM R could have advantages there, although not necessarily so. It is only fair to point out that these figures do not indicate the overall performance of SYSTEM R, which depends on many factors including query optimisation.

In the earlier versions of the Codasyl model [8] records are given their database keys (surrogates) in four Location modes: Calc, Direct, Via and System-default. In most implementations, database keys contain some physical locations such as area number, page number, page offset etc., despite some more recent claims on the independence of database keys from the physical storage. The Location modes are used to generate database keys.

The Calc mode employs hashing on a user-defined Calc key to produce database keys. Random access by the Calc key is fast, taking about 1.33 disc accesses for 30 percent overflows (IDMS, IDS-II), as in our case shown in Table 1. Since the usual practice is to employ the remainder hashing, it is difficult to see how these records can be accessed sequentially by the same key with any efficiency. The storage wastage is implementation-dependent, but usually high, averaging around 30 percent.

In the Direct mode, the database key is taken as the value of a user-defined Direct key. In the event of a clash with an existing database key the system allocates a different database key; if the user forgets this database key later, then it is his problem. These records can be accessed randomly fast by the Direct key except where there are those clashes; efficient sequential access by the same Direct key should be possible, but can not be guaranteed. Note that there is no concept of primary or unique key in the Codasyl model, and if the values of any Calc or Direct key change later, due to updates, then those database keys can not be found.

In the Via mode the database keys are allocated in such a way, that the records concerned are stored close to those of another record type (set type). This mode permits these records to be accessed fast, in some sequence, in association with the set owner. In the last mode, the user has no control over the database keys and hence over the storage locations of these records. Here the notion of fast access by any particular key does not exist in this case except by user-defined indexes.

In the Codasyl model, storage reorganisation will generally be difficult, irrespective of implementors.

5. CONCLUSION

The surrogate implementation technique presented above provides a fast random and sequential access in primary key order in spite of unordered insertions and deletions, with under 10% storage wastage. Access by secondary keys are not adversley affected, and a large measure of storage independence is provided. The technique is implemented in the PRECI [9,10] database system which is based on a canonical data model capable of supporting relational, Codasyl and other user views.

A critical factor of this method is the nature of the pkey value distribution. The method works best if the distribution is reasonably uniform, or can be made uniform (by the key compression algorithm). The overflow mechanism cushions against some non-uniformity. We looked at some actual key values: product codes of a manufacturing concern, and the student numbers of an institution; but both turned out to be rather simple. As indicated earlier we have also developed a generalised key compression technique capable of dealing with most non-uniform distribution reasonably well [11]. Note that we have in fact presented a two-part technique, the parts can be used independently of each other:

(i) Surrogate implementation. Its PINDEX can be a B-tree. Indeed in the PRECI implementation, we have an option of specifying a B-tree instead of a hash tree for the PINDEX of a relation if B-tree is expected to be more efficient for that relation.

(ii) Hash-tree. If the depth of a B-tree is greater than two, then hash tree could be a better alternative for unique-key indexes and probably for non-unique-key indexes. In many machines the basic page size (as unit of input-output) is much smaller than 4096 bytes used in SYSTEM R. For instance in our machine (Honeywell 66/80) it is 1280 bytes, requiring a three-level B-tree for more than 4096 tuples (assuming the key and TID sizes to be the same as those used in the SYSTEM R comparison made earlier).

We intend to examine our technique further in the following areas:

(i) Very highly clustered distribution

(ii) Handling of variable length tuples

(iii) Sharing of data pages by tuples of different relations

(iv) Reorganisation of surrogates for improved performance

(v) Use of hash-tree for non-unique secondary key indexes (currently being studied and implemented).

Finally I would like to thank some of my colleagues in the PRECI project for comments and suggestions - they are: David Bell of Ulster Polytechnic, and Talib Abbod, John Edgar, Dirk Nikodem, Malcolm Taylor and Ambrish Vashishta of Aberdeen University. Many thanks also to Professor D. Kerridge of Statistics at Aberdeen University for assisting me with the probability calculation. The work is partially supported by the U.K. Science and Engineering Research Council.

REFERENCES

1. Hall, P. et al. : "Relations and Entities", Modelling in DBMS, edited by Nijssen (North-Holland 1976).

2. Codd, E. F. : "Extending database relational model to capture more meaning", ACM TODS, vol (4:4), p397, December 1979.

3. Comer, D. : "The ubiquitous B-tree", ACM Computing Surveys, Vol 11, no.2, p121, June 1979.

4. Litwin, W. : "Trie Hashing", Proc. of ACM-SIGMOD, 1981.

5. (i) Astrahan, et al. : "SYSTEM R: Relational Approach to Database Management". ACM TODS, Vol.1, 1976.

   (ii) Astrahan et al. : "A history and evaluation of SYSTEM R", RJ2843 (36129) IBM Research Laboratory, San Jose, California, June 1980.

6. Selinger, P.G. et al. : "Access path selection in a relational DBMS", RJ2429, IBM Research Laboratory, San Jose, California, August 1979.

7. Blasgen, M. W. et al. : "SYSTEM R: An architectural update". RJ2581 (33481). IBM Research Laboratory, San Jose, California, July 1979.

8. Codasyl DDLC Journal of Development, 1978.

9. Deen, S.M., Nikodem, D., Vashishta, A. : "The design of a canonical database system (PRECI)", The Computer Journal, vol(24:3), p200, August 1981.

10. Deen, S. M., Edgar, J.A., Nikodem, D. and Vashishta, A. : "Run-time management in a canonical DBMS: (PRECI)", proc. of 2nd British National Confc. on Databases, Bristol, July 1982, edited by Deen, S.M. and Hammersley, P.

11. Bell, D. A. and Deen, S.M. : "Key space compression and hashing in PRECI" Computer Journal (in press), 1982.