INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • Phone (510) 643-9153 • FAX (510) 643-7684

Reorganization in Persistent Object Stores

Reda Salama
Al-Azhar University
Computer and Systems Dept.
Naser City, Cairo, Egypt

Lutz Wegner[1] and Jens Thamm
Universität Gesamthochschule Kassel
FB Mathematik/Informatik
D-34109 Kassel, Germany

TR-097-023

May 1997

Abstract: The Record Identifier (RID) storage concept was initially made popular through IBM's System R. It remains in use for DEC's Rdb and IBM's DB2 and is attractive because of its self-contained nature. It can even be combined with pointer swizzling. Although simple in principle, its details are tricky and little has been released to the public. One particular problem is the reclamation of empty space when a RID-file becomes sparsely populated. Since RIDs, also called Tuple Identifiers (TIDs), are invariant by definition, pages can be deleted physically, but not logically. Therefore, there must be a mapping from "old" to "new" page numbers. If the self-contained nature is to be preserved, this is not to be achieved by a table but rather through some arithmetical "folding" similar to hashing schemes. Page numbers are meant to collide creating merged pages. The paper explains in detail an efficient division-folding method where f adjacent pages are merged into one. This process can be iterated should the load factor of the file continue to decrease. The algorithm is also designed in a way that the reorganization can be done on-line and step-wise whenever the system is idle. The paper closes with empirical data concerning the relationship between expected number of "spill pages" (extra pages resulting from an overflow in the merge step) and load factor before and after reorganization.

Keywords: persistent storage, file reorganizations, pointer swizzling, complex objects

---

1. Paper completed while L. Wegner was a visitor to ICSI, Berkeley in May 1997.

1 Problem Definition

The Storage Management Layer (SML) of a database system, either relational or object-oriented, associates primary keys, surrogate keys or other identifiers with objects, stores these objects on disks, retrieves them upon request, and performs a number of other tasks. One essential requirement is that object identifiers remain unchanged (invariant) because they are stored in several places, e.g. base tables, materialized views, indexes, etc. Another requirement is that the objects themselves can grow or shrink in size which implies their movement to different storage locations.

There are basically two different techniques for handling the conflicting requirements of invariant addresses and movement of objects:

- address tables, primary B^+-trees or a similar index and
- forwarding schemes.

In the first solution, the identifier is a logical identifier or key, often called Object Identifier (OID) or Database Key. It serves as an index into a table or guides the search in a tree index which contains the actual disk storage address. Another table might exist which maps objects to main memory addresses (pointer addresses). Whenever an object has to move because it doesn't fit into its old location, the new address is entered into the table(s) or primary tree. Inter-object reference is by means of OIDs or primary keys. In particular, OIDs never change and are usually never reused once an object has been deleted. A well-known commercial relational DBMS which uses this tabular approach for object movement is ADABAS [6, p. 763]. For an object-oriented DBMS it is a quite common technique, as e.g. in ORION [5]. According to [9], primary B^+-trees are e.g. Tandem's default organization.

Obviously this tabular or index approach offers great flexibility. Its main disadvantage is the need for large tables or indexes which must be stored on disk as well and must be synchronized for concurrent access. Quite often, the OID-address space becomes so large (2^{64} typically, some people even suggest 128-bit OIDs) that OIDs are first hashed onto some smaller index which is used to retrieve the actual address (see also [4] for different techniques).

In the other approach, the identifier is a physical address, usually called a Tuple Identifier (TID) or Record Identifier (RID). This RID directly specifies a page and some offset into the page, usually with indirection by means of a small table inside this page to allow for shifts within the page. When an object grows and cannot be accommodated inside its present page, it is moved to another page and the new address (forwarding address) is stored in the original location. Should another movement be needed, the object moves to yet another page, but the original forwarding address is changed simultaneously. This way no more than two page fetches are required to access an object. This technique was made popular with IBM's System R [1]. It is in use today by the well-known storage system WiSS [3] as used e.g. in the OO-DBMS O_2 [5], and in both DEC's Rdb and IBM's DB2 according to [9].

The advantage of this approach is that the data are self-contained and that they are accessed directly, i.e. there is no need to store another table or index-tree which often requires a second disk access, must be flushed to disk at checkpointing times, needs extra locks, etc.

However, the RID-approach has some subtle problems. One is that after a set of pages has grown to a certain size and a RID *R*, say *R*=(*p*, *s*) has been assigned, where *p* is the page number and *s* a slot index, the file cannot be truncated to less than *p*+1 pages (assuming numbering of pages 0, 1, ..., *p*) even if all or most of the pages before *p* are empty. Paged files of this type arise in modern multimedia systems where large data chunks come from multiple streams (e.g. monitoring systems, multi-channel operating systems [8]) and are temporarily stored in tables and then filtered which deletes all irrelevant entries.

Here we present solutions for storage compaction in RID-files when these files become sparsely populated due to a large number of deletions. The solution we offer uses what we call *folding of pages* and causes almost no overhead both in access times and extra storage requirements.

The remainder of the paper is organized as follows. In Section 2 we list general requirements for reorganization and describe the usual RID-implementation as known from System R and implemented e.g. in ESCHER [11]. Section 3 then explains the new techniques of folding and lists the algorithms both for folding and for object access. Section 4 gives empirical evidence of the feasibility and inherent limitations of our approach. Section 5 summarizes the results and mentions open problems.

2 Folding a RID-file

2.1 Requirements for Effective Reorganization

A RID-file grows by having new objects added to it or when existing objects are increased in size. Depending on the allocation policy of the SML, growth might for some time be achieved by filling existing pages to maximum capacity. Past some point, however, new pages must be added, at least logically, at the end of the file.

Similarly, a RID-file shrinks when objects are deleted or are reduced in size. If pages at the end become totally empty, they can be truncated. If, however, at least one object remains inside a page, that page and all preceding pages cannot be truncated.

To judge the state of a file with respect to the amount of data it holds, we define the *load factor* of page *p*, short q_p, $0 < q_p \leq 1$, as total amount of data in page *p* / page size. Unless otherwise specified, let the load factor also include meta data, like headers and slot lists . For a file *F* with *N* pages we then let $0 < Q \leq 1$ denote the *file load factor,*

$$Q = \frac{1}{N} \sum_{i=0}^{N-1} q_i$$

From hash file organizations we know that efficient operation can be expected when load factors do not exceed about 0.85. In B- and B*-tree organizations, load factors of at least 0.5 are guaranteed and around 0.65 are to be expected on the average [6].

For RID-files, no direct correlation exists between load factors and performance, at least in static files. In the dynamic case, when frequent insertions/deletions/size changes happen, there is an indirect correlation in that clustering deteriorates and displacements increase when the load factor is high. Here, however, we are mostly concerned with situations where the load factor drops below a certain threshold, typically well below 50% utilization ($Q < 0.5$). This is a common phenomenon in database systems where a table represents a queue, say of applications or messages, which first grows up to a large size and then drops again once most queued items have been handled and leave the queue.

A reorganization of persistent storage for these type of applications must fulfill several requirements:

- storage utilization after reorganization should be better than 50% and preferably be in the 80 to 90% range;
- the overhead introduced for handling access after the reorganization should be small and should possibly cancel against the savings from having smaller files;
- therefore access time to data before and after the reorganization should not differ by more than a small factor, say 1.1 or 1.2 (10 to 20% increase in access time after reorganization);
- reorganization should occur on-line and step-wise whenever there is low activity; it should not disrupt operations on the file;
- it must be possible to halt storage compaction at any time and even to reverse the process, in case a period of growth is encountered;
- reorganization principles should be simple and generic so that the algorithm works regardless of implementation specific details.

2.2 Review of the RID-design

To judge the following reorganization scheme, it is necessary to first understand the RID addressing principle. Consider the following Figure 1 with two pages and several objects stored inside these pages.

Page *p* contains 4 stored objects. Two of these objects are true system objects (meta objects which belong to the system), namely the header and the free space. Slot 0 which holds byte address 4 thus points to the start of the header. Direct manipulation of the header is forbidden and there is a set of dedicated routines to handle header contents which is omitted here.

The free space in page *p* starts at byte address 1028 and extends to the start of the slotlist which is address 4086 (page size 4096 - 5*2 bytes for five slots of size 2). This address is also seen in a 2-byte field at the very beginning of the page.

The other two “normal” objects are a 1,000-byte object in slot 1 and a *displacement* RID in slot

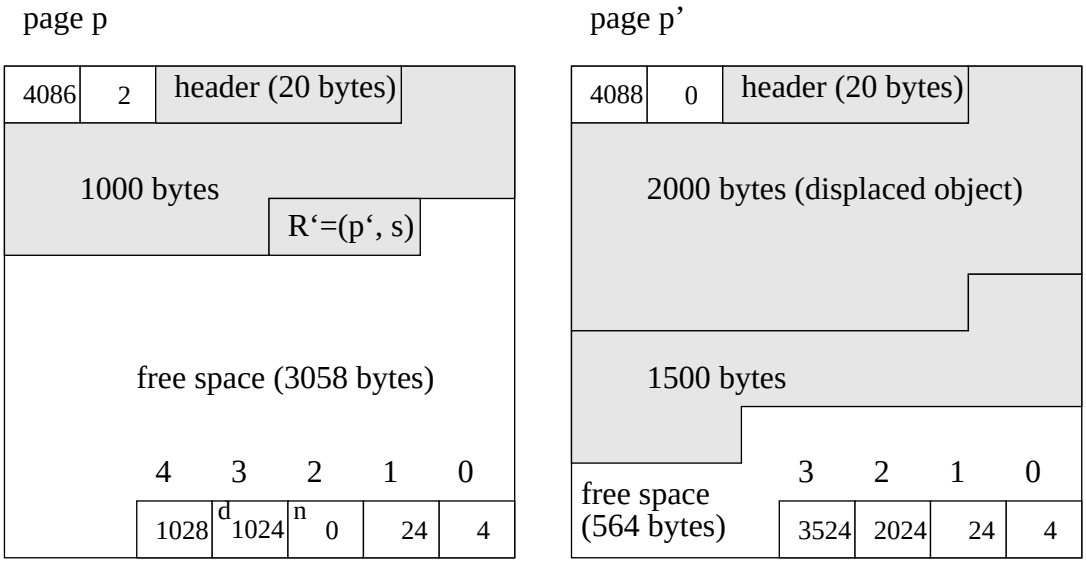


Figure 1: Two example pages

3. Displacement RID means that the actual object is on another page, here *p*’, and its new RID *R*’ is stored in the original (invariant) location. The reason for dislocating the object from page *p* could have been insufficient space at the time when the object grew.

Finally, pages can contain empty slots corresponding to deleted objects. Empty slots can be recycled[1]. Slots marked as empty form a linked list with the last entry having value zero. In our example, page *p* contains one empty slot, namely slot 2, and the second field in the start of the page contains the index of this (first) slot in the chain of empty slots. Note also that object sizes are determined by subtracting an object’s starting address from the starting address in the next higher non-empty slot.

Page *p*’ is quite similar but does not have an empty slot. Note also that the displaced object is not marked in any way, i.e. one cannot determine that any object in page *p*’ is a displaced one by looking at *p*’ alone. In some implementations, the displaced object is marked as a “protected object”, just like the header and the free space, to catch any erroneous access to this object[2].

Even if the displaced object where marked as such one would need to store the original RID (a “back pointer”) in order to freely reorganize pages. We will return to this severe limitation of the RID-design later on.

---

1. If recycling of RIDs is undesired, objects should be truncated to size 0 or changed into small tombstones.
2. Remember: if (p’, s’) is the RID of a displaced object with original RID (p, s), then (p’, s’) should not be released to the outside world because the object might migrate again; only (p, s) is guaranteed to be invariant and any access is only via this RID.

For the following discussion we also keep to the usual RID-file parameters which assume 4, sometimes 8 KB pages. Slots are normally 16 bits wide which allows 12 to 13 bits for the address inside the page (8 KB = 2^{13}) and 3 bits for flags (normal, displaced, empty, protected, ... object).

3 Remapping of Page Numbers

3.1 Division versus Modulo Method

Assume a RID-file has grown to a size where it occupies N pages, i.e. the highest index of a page (page number) is N-1 given that the first one is numbered 0. Then, once an object has been allocated with a RID $R = (N\text{-}1, s)$ for some $s > 0$, and once that invariant RID has been handed to an application, the file cannot be truncated to a size less than N without explicit or implicit remapping of page numbers.

One method would be through the use of a mapping table. This, however, leads to the address (database key) approach mentioned earlier and is not considered here because it does not preserve the self-containing nature of RID-files.

The other method is to convert all page numbers by some form of arithmetic into page numbers of a smaller range. This is similar to hashing except that we want a high and evenly distributed number of collisions which represent “merged” pages. Here, merging of pages implies some way of throwing objects from two or more pages into one single page freeing the other pages for reclamation or other use.

We consider two straight-forward mappings.

- division folding: $p' = p$ **div** f
- modulo folding: $p' = p$ **mod** $\lceil N/f \rceil$

where p is a given page number, p' the newly computed page number, f is a folding factor, and $\lceil x \rceil$ denotes the smallest integer which is greater or equal to x. To understand the differences, consider Figures 2 and 3 with f set to 2.

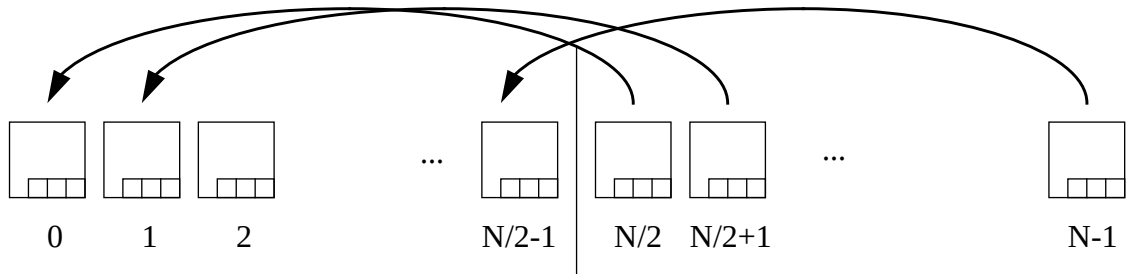


Figure 2: Mapping with modulo-method

With $f = 2$, a 2-way fold is performed, i.e. the file will be folded once which reduces its size to half, provided all pages can be merged. In the **modulo method**, pages which are N/f apart are merged[1], i.e. 0 goes with $N/2$, 1 with $N/2+1$, etc. Clustering is not affected by this method apart from a jump of locality around the folding point $N/2$. If $d = \lceil N/f \rceil$ is a power of two, say 2^k, then this method corresponds to keeping the k lowest bits to determine the new page number p'. Furthermore, p **div** d tells us always from which side of the file the page came from: in the case of $f = 2$, 0 is left side, 1 right side. Similarly for higher values of f, p **div** $\lceil N/f \rceil$ yields 0, 1, ..., f-1 for the first, second, up to the f'th stretch.

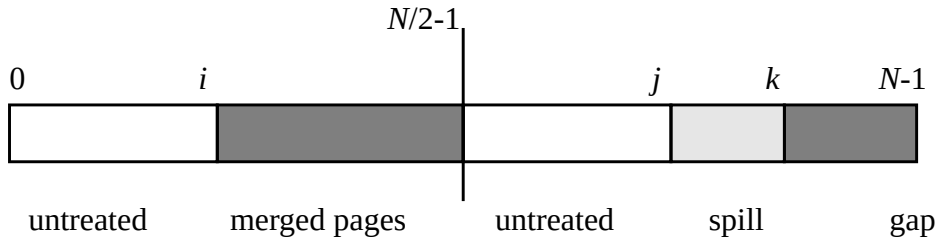


Figure 3: Folding for modulo method from right to left

For the actual merging process consider Figure 3 which depicts again the case $f = 2$. The process of merging pages is from the right to the left starting with pages $j = N$-1 and $i = N/2$-1. They are loaded, merged and rewritten to location $N/2$-1. Then i and j are decremented. Figure 3 also shows a shaded area between j+1 and k, called the "spill", which leads into the consideration of how to handle pages which cannot be merged.

Since individual pages can carry any perceivable load from 0 to `MAXOBJECTSIZE`, it can happen that the f pages to be merged don't fit into a single page, although the file as a whole might have a load factor of less than $1/f$. This results in what we call *spill page*s which come to rest outside the merged area but inside the file. In Figure 3 they form the lightly shaded part between j+1 and k. They must be moved to the left as merging progresses. This will be done by a technique called the wheel which was introduced in [10] and requires the exchange of one page per move (page k is moved to location j). Details of this handling of spill pages are skipped here and are reconsidered in the context of the detailed explanation of the division-algorithm below.

In Figures 2 and 3 above, folding was depicted with $f = 2$ which we might call **binary folding**. For the modulo-method, folding with $f > 2$, called **multi-way folding**, exhibits some unpleasant effects.

Firstly, multi-way folding creates multiple gaps of freed pages. Those "inside" the file cannot be truncated. Those at the tail end of the file can be truncated, but up to the last step, only up to N/f - 1 pages are returned to the system.

1. For the example with $f = 2$ assume without loss of generality that N is even.

Secondly, it may even happen that in the immediate steps the file actually grows (!) because spill pages are created at the tail end while simultaneously gaps of empty pages appear inside the file. Of course, spill pages could be differently chained to stay inside the gaps but that would require keeping track which pages are spill pages and which are freed which is not very attractive.

Thirdly, at the end of folding, the spill pages as a block must be copied back to the tail end of the file which has been reduced to size N/f (+ spill).

Folding a file with the modulo method suffers from another few, very subtle, defects, e.g.

- handling repeated folding when the file size was not a power of 2
- merged slot lists inside pages become interleaved.

This finally made us decide to drop the modulo algorithm and to consider the division-method instead.

3.2 The Division-Folding Algorithm

In the **division method**, f adjacent pages are merged, i.e. in the case of $f = 2$, pages 0 and 1 become page 0, 2 and 3 become 1, 4 and 5 become 2, etc. as shown in Figure 4. If f is a power of 2, the new address can be thought of as deleting the lowest log f bits which can also be very efficiently achieved by shifting log f bits to the right during the address calculations. Furthermore, clustering increases with every folding.

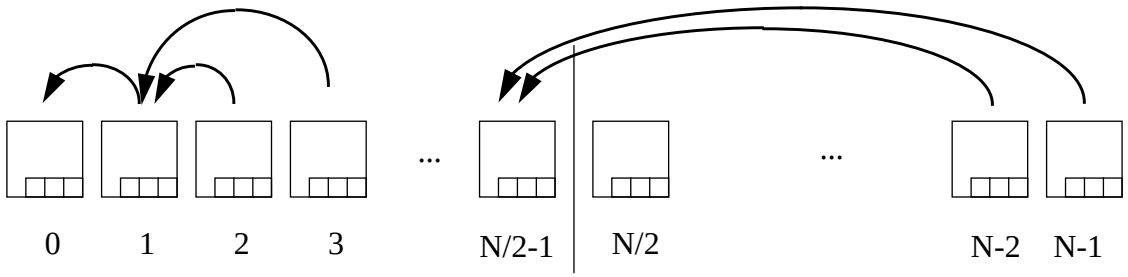


Figure 4: Mapping with division-method

On the other hand, the folding process is most naturally performed from left to right creating a larger and larger gap inside the file (see Figure 5 and explanation in next paragraph) and requires loading three pages (in general $f + 1$ pages) for each merge step except the first one. Although there are ways to improve this scheme, e.g. with some clever buffering scheme and doing up to $\lceil \log N \rceil$ merges in one step starting at the right, the left to right method is much simpler. The disadvantage is that no pages are returned to the system until the end of the reorganization pass.

Consider now the details of the left to right division folding method. The f pages which are merged in each iteration are at locations k, k+1, ..., $k + f$ - 1, initially starting with $k = 0$. The

result goes into page $i = k$ div f (initially $i = 0$), i is increased and k increases by f which is also the increase in the gap of freed pages. Again, a wheel of spill pages is maintained between i and j-1 and is rolled to the right at each folding step.

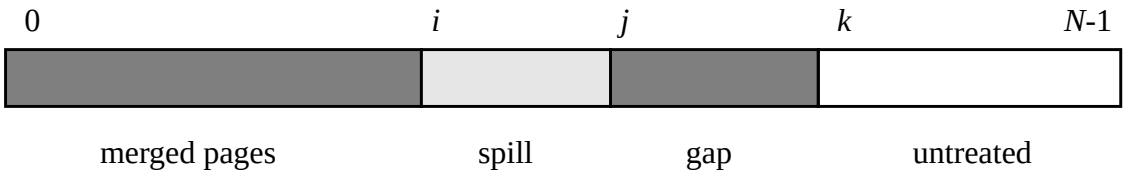


Figure 5: Folding for division method from left to right

The process stops when k reaches N-1 and in the last step less than f pages might be merged if N mod $f \neq 0$. The resulting file has size $d = \lceil N/f \rceil$ plus whatever spill pages were produced.

This leads to the following straightforward algorithm written in Pseudo-Pascal, where we assume that the folding factor f is decided beforehand based on a measured load factor Q.

```
procedure FoldFile( file F; int f, N);
var
    i, j, k: int;
begin
    i := j := k := 0;
    while idle and (k < N) do
        foldstep(F, N, f, i, j, k);
end {FoldFile};

procedure foldstep(file F; int N, f; var int i, j, k);
{create a gap of one page at location i and merge f pages starting at location k
    into this new location}
begin
    RollWheelUp(F, f, N, i, j, k);
    i := i + 1; j := j + 1;
    if EnoughSpace(F, f, k)
    then begin
        MergePages(F, f, N, i, j, k);
    end
    else begin
        MergeAndHandleSpill(F, f, N, i, j, k);
    end;
end {foldstep}

procedure MergePages(file F; var int f, N, i, j, k);
var page: int;
    buf_a, buf_b: buffer_for_page;
```

```
begin
    LoadPage(F, k, buf_a);
    k := k + 1; page := 1;
    while (k < N) and (page <= f - 1) do
    begin
        LoadPage(F, k, buf_b);
        MoveToPage(F, i-1, k, buf_a, buf_b);
        MarkAsEmpty(F, k);
        k := k + 1; page := page + 1
    end {while};
    WriteBackPage(F, i - 1, buf_a)
end {MergePages};
```

Let *MoveToPage*(file F; int a, b; buffer_for_page buffer_a, buffer_b) be a procedure which moves the contents of page no. *b* into page no. *a*, where *a* is the number of the final destination of the merged pages. Figure 6 shows the logical scheme. The procedure assumes that both

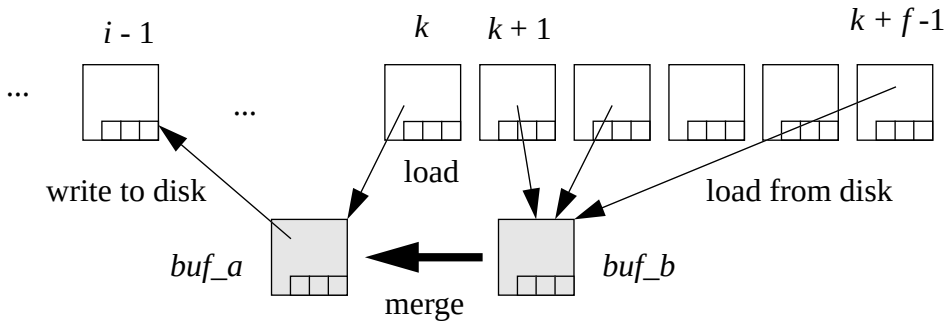
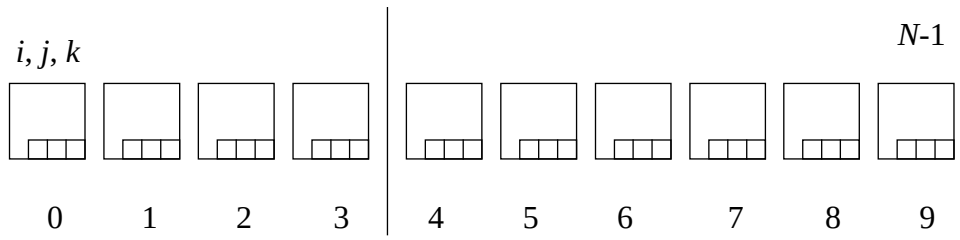


Figure 6: Folding f pages via two buffers

buffers *buffer_a* and *buffer_b* have been loaded before. Afterwards, page *b* will be logically empty. Details of *MoveToPage* are given below in Subsection 3.4.

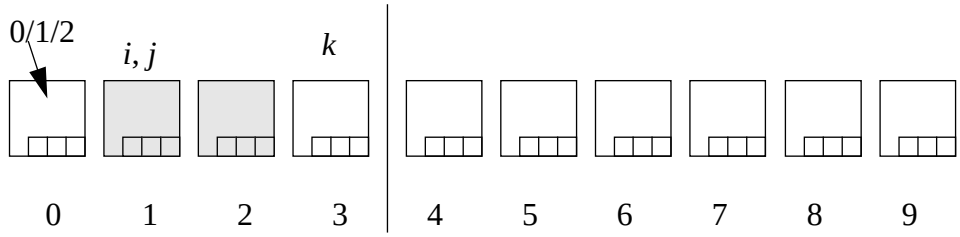
Example 1

Consider the three-way ($f = 3$) folding of $N = 10$ pages. The line indicates the final length $d = \lceil N/f \rceil$, here $\lceil 10/3 \rceil = 4$.



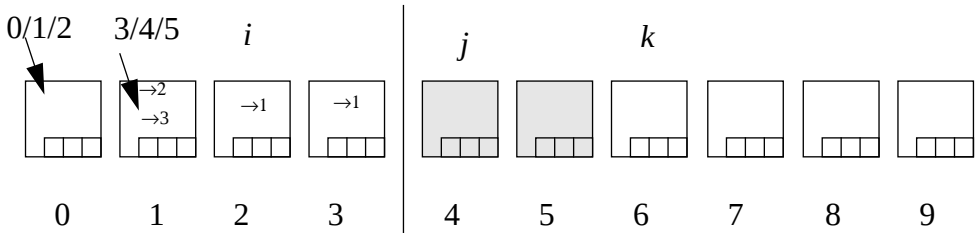
Step 1:

In the first round, pages 0, 1, and 2 are folded. They are merged into page 0 creating a gap of two pages, depicted as shaded squares. There is no wheel of spill pages.



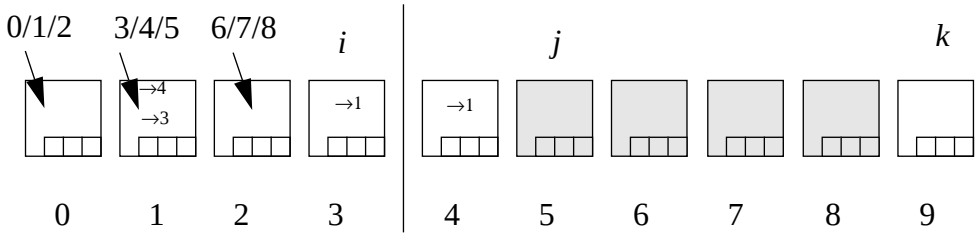
Step 2:

In step 2, pages 3, 4 and 5 are merged. Assume, all three won't fit into one page and that 4 and 5 both become spill pages. Page 1 (previously no. 3) holds two anchors which indicate that pages 2 and 3 hold a wheel of spill pages. Note also, that these spill pages contain "back references", here indicated with "→1".



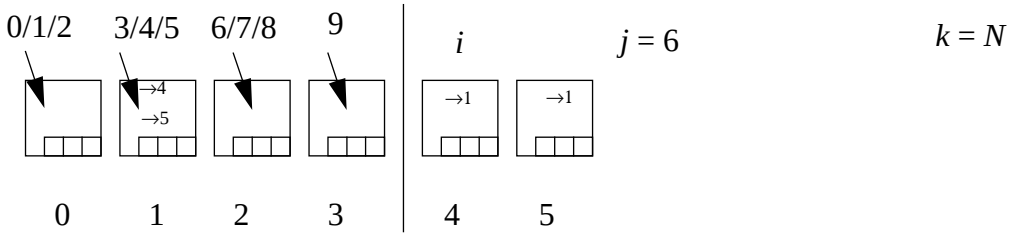
Step 3:

In the third step, we assume all three pages 6, 7, and 8 fit into one page. The wheel is rolled once to the right causing adjustments of one anchor in page 1.



Step 4:

In the final step, page 9 becomes page 3 and the wheel is again rolled once with the usual



adjustment of one anchor. This leads to the final configuration with pages 6 - 9 truncated. ■

We will use this example to discuss some design alternatives.

3.3 Merged Pages

The final crucial point is the design of the merging of pages. Here it is important to remember that empty slots can only be removed if their index is higher than the highest non-empty slot. We will show in Section 4 that, if pages have an average slot list length of S slots at the time of their maximal growth ($Q = 1$) and if subsequently the file load shrinks to a factor $Q << 1$, slot lists will reduce their length by at most $(1 - Q)/Q$, i.e. the reduction is independent of the list length S! This creates a phenomenon very similar to the file itself: although the page could become almost empty, the slot list cannot be truncated past the highest occupied slot.

Example 2

If 4 KB pages are considered and object sizes vary uniformly between 1 and 100 bytes, a page can take an average of 80 objects, i.e. $S = 80$. If now objects are randomly deleted down to a load factor of $Q = 0.1$ (90% deleted), the new average slot list length is $S' = S - 0.9/0.1 = 80 - 9 = 71$. If the file were compacted five-fold times to bring Q back to 0.5, we merge $f = 5$ pages into one which implies having, on the average, to deal with $5 * 71 = 355$ slots or about 0.7 KB of metadata. ■

In the example above, the large amount of slots makes it unlikely that $Q > 0.75$ can be achieved unless there is a way to also merge the large sparse slot lists. However, the example might be misleading. As our empirical tests indicate, slot list lengths depend very much upon the application. If a 4 KB page holds objects of 500 bytes average size and subsequently looses almost all objects ($Q = 0.1$), we observe average slot lists between two and three and even for f = 8, throwing all slot lists together only requires less than 50 bytes! Similarly, $Q = 0.1$ might also be, on the average, a file with one full page versus nine empty ones for every ten pages considered.

Common ways to handle sparse vectors include bit vectors to indicate non-empty entries, condensed sequential of chained storage of pairs (slot-index, slot-value), hashing, etc. In most cases, however, we loose direct access to the slot value which, in particular in connection with software based pointer swizzling [11], seems unacceptable.

One interesting alternative which we considered was collision-free folding of slot lists as indicated in Figure 7. There the slot lists, say X of page p_x and Y of page p_y, are represented as bit-vectors with a 1 representing a non-empty slot. Using bit-wise **AND**, we shift vector Y one bit to the left filling up with zeroes (logical shift left) as long as X **AND** $Y \neq 0$. When after k shifts (ultimately in the worst case after $k = n+1$ shifts for n = length of X) no conflicts occur, we can save the starting position k and access any slot in the combined slot vector, say XY, with $XY[s']$, where slot index $s' = s$ if $R = (p_x, s)$ and with slot index $s' = s + k$ if $R = (p_y, s)$.

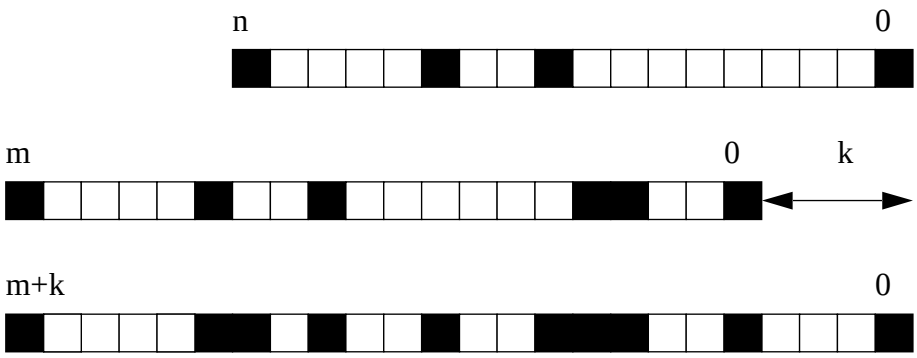


Figure 7: Collision-free folding of slot lists

However, this method has several disadvantages. Firstly, the probabilities for successfully folding two (or even more) lists is small. Secondly, the SML cannot catch invalid RIDs because it cannot distinguish between slot entries from different pages. For the same reason it cannot unfold pages in the case where the file grows again.

Therefore, the design decision was to simply concatenate existing slot lists from the merged pages and to place a little table into the available free space. This table with *f* (the folding factor) entries contains the start addresses of each slot list (see Figure 8).

Figure 8 assumes the merging of three pages. The first page contained a header, a 1000-bytes object, an empty slot, a displaced object. The slot for the original free space is deleted. As for the second page, it had originally a header, a 500-bytes and a 100-bytes object. Again the slot for the free space is deleted.

The third page had header, an empty slot, a 1100-bytes object and, finally, the slot for the empty space of the whole page. This empty space amounts to 4096 - (2768 + 22 + 6) = 1300 bytes, where 22 bytes account for the slot list of 11 slots and the little table has f elements with a two byte address each.

Naturally, details may vary with implementation specific requirements, e.g. whether headers are copied into the merged page or not. The important point, however, is that merged data go behind existing data objects and that slot lists go in front of existing slot lists. Thus, there is no need for shifting data inside the page.

Also the calculation of object sizes remains unchanged. Finally, when folding of the file continues in additional rounds, the merged slot lists and tables are easily added without shifts.

3.4 Calculating Object Adresses at Run-Time

What remains to be shown is the calculation of an object addresses A for a given RID R = (p, s). Clearly, the proper conversion of page number and slot index depends on the folding his-

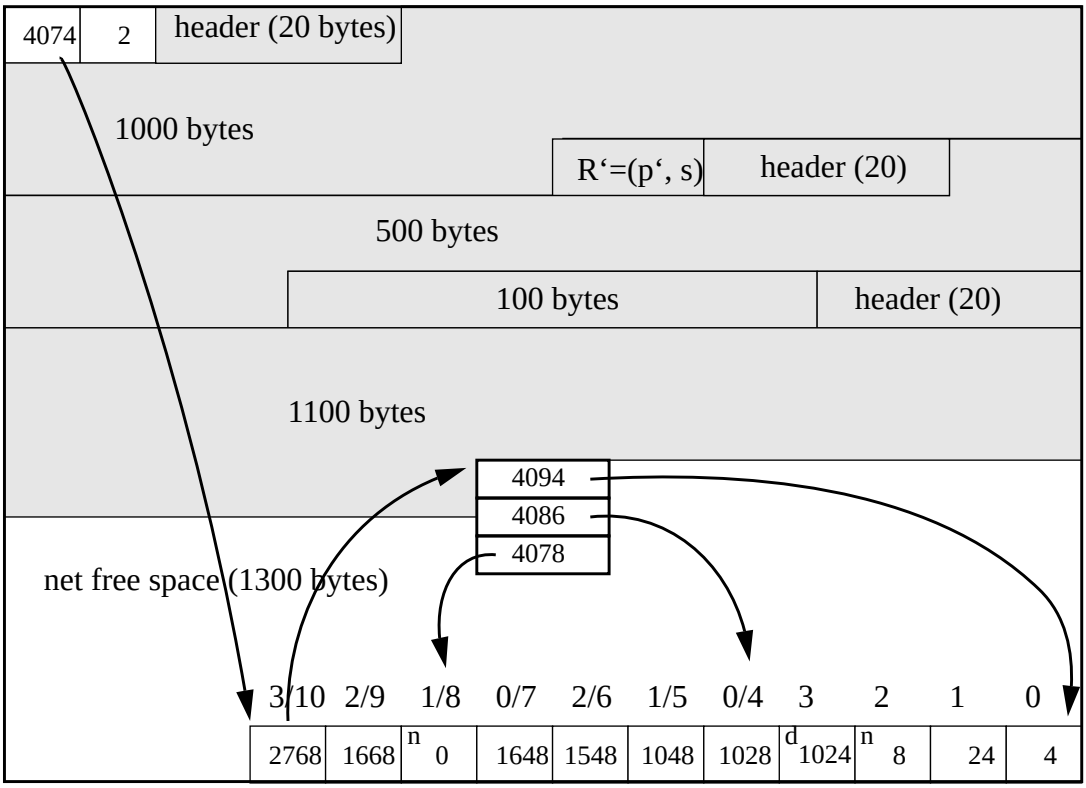


Figure 8: Merged page with *f* = 3 slot lists

tory. In particular, it must take into account that folding might be in progress.

Fortunately, this requires only three integers which the system must remember:

- old folding factor *f_old* (initially 1)
- new folding factor *f* (initially 1)
- start page *k* of the untreated portion.

For the algorithm which follows, assume that *Slots* is a variable which looks at the 4 KB page as an array[0 .. 2047] of 16-bit addresses and let *Page* be the array[0 .. 4095] of bytes (char). Note also that the 16-bit values inside the slots—unless they are empty slots—themselves are byte addresses which must be divided by 2 to convert to slot numbers.

p’ := *p* **div** *f_old*;
if *p*’ >= *k* {*in the untreated section of the current folding process*}
then *proper_f* := *f_old*
else begin *p*’ := *p* **div** *f*; *proper_f* := *f* **end**;
load *p*’ unless already in main memory,
let *PA* be the starting address of the buffer for this page
SlotContent := *Slots*(*PA*^.*Page*)[*Slots*(*PA*^.*Page*)[*Slots*(*PA*^.*Page*)[

Slots(*PA*^.*Page*)[0] **div** 2] **div** 2 + *p* **mod** *proper_f*] **div** 2 - *s*];
if *SlotContent* > $3FFF
then ... displaced or empty object or object in spill page ...
else *A* := Addr(*PA*^.*Page*[*SlotContent* **and** $1FFF]);

Although this address calculation seems pretty expensive, it actually is not. Compared to the original RID-access algorithm, the cost of the access algorithm has increased by one comparison (*p*' >= *k*), a division operation (*p* **div** f_*old*), a modulo operation (*p* mod *proper_f*) and a triple indirection which is cheap. If *f* is a power of two, division and modulo arithmetic can be efficiently done with register shifts.

3.5 Handling Spill Pages

The situation that pages cannot be merged arises most likely when pages contain few, but large objects and two or more of these pages happen to fold onto the same target page. They create what we call spill pages which are stored at the tail end of the file.

Naturally, there are many options. Since RID-files support object relocation, one could try to move a conflicting object individually to a less crowded page. Secondly, there is a choice of moving all or only some of the conflicting objects to a special spill page and, thirdly, there is the choice to treat the page as a whole like a displaced page.

The last choice could easily be accommodated with the little table of *f* entries which we placed at the beginning of the free space (cf. Figure 8). Here, entries would be 4 bytes to take page numbers (which only require 3 bytes in the usual RID set-up) and a flag inside these entries would indicate whether we have a page number or a slot list address.

Here, we opted for the second solution and move all objects from a conflicting page to one spill page, but the displacement is individually handled as if a displaced object had to be handled. This saves an extra comparison in the address calculation. The disadvantage is that chains of dislocation can extend to 3 pages. Thus, whenever objects are accessed with the ultimate loca tion inside a spill page, objects should be relocated to reduce the access path to at most two page reads.

As for the back pointer inside a spill page, we suggest storing this page number (i.e. the num ber where this spill page originated from) inside the first 4 bytes of the free space. Should such a page be filled to the very limit with free space of zero bytes, we can adopt the convention that linking of empty slots is not done for spill pages which frees two bytes in bytes 2 and 3 of the page (see Figure 8, the entry in the left corner containing the slot index 2). Similarly, the header always starts at byte position 4, so this value is redundant and could be used to create the needed page number. Other possibilities are fields inside the header which are not needed for spill pages.

In general, these finer details are design dependent and are best solved by leaving extra space in pages to start out with. It is also clear that spill pages do not participate in (future) folding

steps. However, as further objects are deleted, pages could become empty and may then be deleted altogether. Since each spill page is linked to its originating page, the resulting gap can be closed with a page from the end and the file truncated accordingly.

4 Empirical Results

The proposed folding algorithm invites all kinds of empirical investigations regarding optimal folding factors and starting times, expected improvement of storage utilization, etc. Here we concentrate on describing our experimental set-up, the basic assumptions and the conclusions gained from the data.

To study the behaviour of the folding algorithm we created a parametric simulation environment. To run the experiments, the user specifies

- the initial size *N* of the file (number of 4KB pages)
- a distribution for the object sizes with categories and minimum and maximum sizes
- a load factor *Q* ($0 \le Q \le 1$) before folding
- a folding factor *f*.

The experiment then runs as follows. The system creates in turn objects with random sizes, where the size is uniformly drawn from the chosen size range. It then tries to locate a page which has sufficient space to store the object. The search starts with the last page moving towards the front. If a page is found, the simulation returns the RID which goes into an array *A*. If none of the existing pages can hold the object, another page is added to the file. The process stops when the N+1st page would be needed. Finally, statistics are collected which report the average slot list length, average gross and net storage utilization (i.e. with and without space needed for internal meta data such as slot lists), etc. Figure 9 shows a typical situation with 20 pages and fairly large objects (avg. 510 bytes). As can be seen, the particular allocation strategy for objects fills pages rather well leading to $Q = 0.9585$.

In the next phase, the system randomly picks RIDs from array *A* and deletes the corresponding objects. Subtracting the freed space from the total space, the system continues until storage utilization drops below *Q* for the first time. The system then reports again the current status (see Figure 9 with *Q* aiming at 0.4).

The most important observation here is that although slot lists contain many free slots (in the example 3.5 slots on the average), the slot list length has been reduced from 7.60 to only 6.60. The reason is that the highest non-deleted RID in a page determines the final list length. In fact we can precisely estimate the slot list reductions when deletions reduce a RID-file to a load factor *Q*.

$$E(Listreduction) = Q \sum_{i=0}^{\infty} i(1-Q)^i = \frac{1-Q}{Q}$$

```
minimal record size: 0 (4)
maximal record size: 1018 (1024)
page count: 20
```

```
page|used|free|#sl|#fs|usage
----+----+----+---+---+-----
   0|3844| 252|  9|  0|93.8
   1|3877| 219|  8|  0|94.7
   2|3904| 192|  7|  0|95.3
   3|3993| 103|  6|  0|97.5
   4|3983| 113| 10|  0|97.2
   5|3973| 123|  6|  0|97.0
   6|4086|  10|  7|  0|99.8
   7|4013|  83| 10|  0|98.0
   8|4011|  85| 10|  0|97.9
   9|3993| 103|  7|  0|97.5
  10|3693| 403|  7|  0|90.2
  11|4061|  35|  9|  0|99.1
  12|3987| 109|  8|  0|97.3
  13|3996| 100|  9|  0|97.6
  14|3546| 550|  7|  0|86.6
  15|3993| 103|  9|  0|97.5
  16|3654| 442|  6|  0|89.2
  17|4095|   1|  8|  0|99.9
  18|3404| 692|  4|  0|83.1
  19|3744| 352|  5|  0|91.4
```

```
record count: 152
total record size: 77546
average record size: 510.17
average free: 203.50
average slot count: 7.60
average free slot count: 0.00
average usage: 95.03
```

```
space quota: 40%
```

```
page|used|free|#sl|#fs|usage
----+----+----+---+---+-----
   0| 952|3144|  8|  5|23.2
   1|1208|2888|  7|  5|29.5
   2| 878|3218|  7|  4|21.4
   3|1459|2637|  6|  4|35.6
   4|2743|1353|  9|  4|67.0
   5| 321|3775|  3|  2| 7.8
   6|1581|2515|  5|  2|38.6
   7|1739|2357| 10|  6|42.5
   8|2827|1269| 10|  4|69.0
   9|2554|1542|  7|  3|62.4
  10| 728|3368|  6|  5|17.8
  11|1036|3060|  7|  3|25.3
  12| 889|3207|  3|  2|21.7
  13|2312|1784|  9|  3|56.4
  14|1454|2642|  5|  2|35.5
  15|1077|3019|  7|  4|26.3
  16|2199|1897|  6|  2|53.7
  17|1250|2846|  8|  6|30.5
  18| 980|3116|  4|  3|23.9
  19|2789|1307|  5|  1|68.1
```

```
record count: 62
total record size: 30712
average record size: 495.35
average free: 2547.20
average slot count: 6.60
average free slot count: 3.50
average usage: 37.81
```

Figure 9: A test set of $N = 20$ pages initially and after reduction to $Q = 0.4$

As the little table shows, the reductions are minimal even in the case of infinite slot lists.

Q	0.1	0.2	0.3	0.4	0.5	0.6
E(reduction)	9	4	2.3	1.5	1	0.6

Figure 9 also shows how page utilization varies strongly with the two random processes (object size and deletion pick) overlapping. If a page has been filled by two large objects and both happen to survive the deletion process, the page can be quite full even if *Q* is below 0.4 (see pages 4, 8, and 19 in Figure 9).

In the third phase, the file is folded with factor *f*. This is done by applying the division-folding

algorithm from Section 3.2 including creation and rolling of a wheel of spill pages as needed. Inside the pages, slot lists are concatenated. The resulting data are again tabulated. Figure 10 shows one particular result for $N = 20$ pages, $f = 2$ and $Q_{old} \leq 0.4$ as shown in Figure 9 before.

```
spill length: 1 page(s)

page|used|free|#sl|#fs|usage
----+----+----+---+---+-----
   0|2166|1930| 16| 10|52.9
   1|2343|1753| 14|  8|57.2
   2|3070|1026| 13|  6|75.0
   3|3326| 770| 16|  8|81.2
   4|2833|1263| 11|  4|69.2
   5|1770|2326| 14|  8|43.2
   6|3207| 889| 13|  5|78.3
   7|2537|1559| 13|  6|61.9
   8|3455| 641| 15|  8|84.4
   9|3775| 321| 10|  4|92.2
  10|2554|1542|  7|  3|62.4

record count: 72
total record size: 30752
average record size: 427.11
average free: 1274.55
average slot count: 12.91
average free slot count: 6.36
average usage: 68.88
```

Figure 10: Result after folding with $f = 2$

In this particular example, only one spill page results, which is caused by the attempted folding of pages 8 and 9. In Figure 10, the two pages become page 4 and page 10. We also note that the new utilization with $Q_{new} = 0.6888$ is well below the anticipated 80% ($Q_{old} * f$).

As it turns out, this particular example is even better than the average which we report below. Some analysis shows that the average usage drops because of spill pages and because of overhead with merged (but mostly empty) slot lists.

To study these effects we ran our simulations with two different data sets. One consisted of large objects whose size was uniformly distributed in the range 0 to 1024 bytes. Figure 10 shows the resulting utilization U (average net usage of space) plotted against initial load factor Q. Since the objects are fairly large and slot lists are short (around eight slots), the difference between gross and net usage is only about 0.5%.

Of course, one cannot expect a high utilization U if pages were initially rather empty (e.g. $Q = 0.1$) and f is low. Therefore, the plot also includes an efficiency factor E which measures how close the actual load factor after folding came to the factor achievable in theory, i.e. $E = Q_{new} / Q_{old} * f$. The upper curve shows E for f = 2 which is well above 0.9 for $Q \leq 0.3$ and at $E = 0.88$ for $Q = 0.4$. With higher loads or higher folding factors, spill pages take their toll.

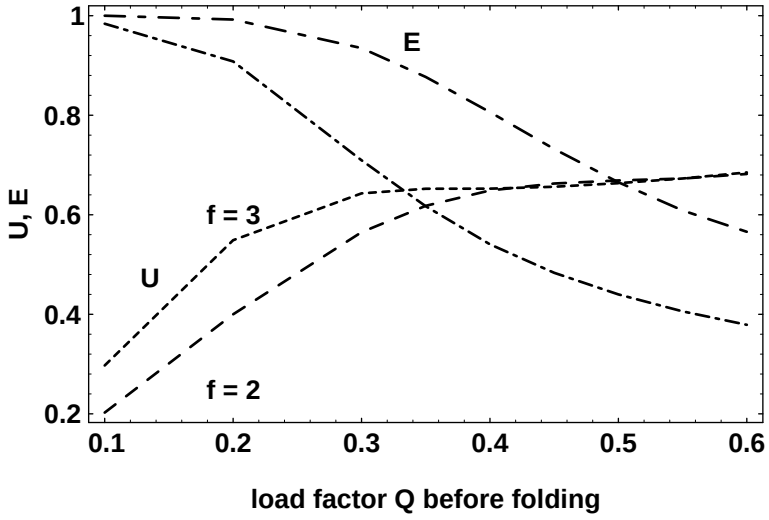


Figure 11: Utilization *U* and efficiency *E* of folding with $f = 2$ and $f = 3$

For the second set we took data from the OO7 benchmark [2] which created four categories of objects as shown in the following Figure 12. Clearly, smaller but numerous objects create more even distributions in pages. However, they also create larger slot lists!

From these data we see that slot lists consume about 6% of the available space in full pages (page size / slot list length * slot size ≈ 4096 / 120 * 2). After deleting 80% of the objects for $Q_{net} = 0.2$, only about 4 slot entries are removed as explained above. Folding this reduced file three times creates 6 spill pages and slot lists of triple length which now amounts to already 14% overhead. At the same time, net usage remains around the 50% level even though one would hope for 3 * Q = 0.6.

When the experiments described above were performed 100 times each with folding factors 2, the curves in Figure 13 result. As before we notice that folding without adjustments for collissions of well-filled pages cannot push gross utilization above the 0.67 mark.

There is also a simple argument for explaining this phenomenon. If the degree of data within a page is random within a certain normal distribution we could classify pages as "*low load*", "*average load*" and "*high load*". When being folded, pairs of *average/average* and *low/high*, *high/low* even out. *Low/low* pairs bring the resulting load factor down and would normally be counterbalanced by *high/high* pairs. The latter however turn into 2 or more spill pages which reduces the load facor even further. Currently, we try to give a more precise statistical explanation which is difficult because we start with random sizes and random deletions.

All in all, the strong influence of randomness in the deletion process together with randomness in the choice of object sizes surprised us. They call for strategies of balancing sizes before merging takes place. This is feasible within the RID-concept but difficult because displaced objects carry no back-pointers with them. No dynamic reorganization technique for RID-files

```
ABC distribution: 4 slices
11940 values
slice| min | max |exp. |real |val #
-----+-----+-----+-----+-----+-----
    0|   18|   18|86.70|86.69|10351
    1|   24|   48|11.56|11.56| 1380
    2|   96|   96| 0.58| 0.59|   70
    3| 1016| 1040| 1.16| 1.16|  139


page|#sl|#fs|used|free|net |gross
----+---+---+----+----+-----+-----
   0|134|  0|4090|   6|93.31|99.85
   1| 92|  0|4090|   6|95.36|99.85
   2|139|  0|4092|   4|93.12|99.90
   3|134|  0|4086|  10|93.21|99.76
   4| 95|  0|4078|  18|94.92|99.56
   5|140|  0|4096|   0|93.16|100.00
           . . .

page count: 100
record count: 11939
total record size: 384786
average record size: 32.23
average free space: 9.36
average slot count: 119.39
average free slot count: 0.00
average net usage: 93.94
average gross usage: 99.77

page|#sl|#fs|used|free|net |gross
----+---+---+----+----+-----+-----
   0|134|109| 746|3350|11.67|18.21
   1| 89| 72| 537|3559| 8.76|13.11
   2|137|100|1062|3034|19.24|25.93
   3|126|105| 644|3452| 9.57|15.72
   4| 94| 68|1693|2403|36.74|41.33
   5|139|110| 865|3231|14.33|21.12
             . . .

page count: 100
record count: 2610
total record size: 81886
average record size: 31.37
average free space: 3046.10
average slot count: 115.52
average free slot count: 89.42
average net usage: 19.99
average gross usage: 25.63

page|#sl|#fs|used|free|net |gross
----+---+---+----+----+-----+-----
   0|361|281|2353|1743|39.82|57.45
   1|360|283|3210| 886|60.79|78.37
   2|310|235|3142| 954|61.57|76.71
   3|354|281|3164| 932|59.96|77.25
   4|358|271|2476|1620|42.97|60.45
   5|361|277|3424| 672|65.97|83.59
           . . .
  39|133|107|1783|2313|37.04|43.53

page count: 40
record count: 2644
total record size: 82090
average record size: 31.05
average free space: 1464.45
average slot count: 289.65
average free slot count: 223.55
average net usage: 50.10
average gross usage: 64.25
```

Figure 12: Data sets of the OO7 benchmark with $Q = 0.2$ and $f = 3$

is currently known in the literature[1].

5 Conclusion

The Record Identifier storage concept was initially made popular through IBM's System R [1]. It is in use today with DEC's Rdb and IBM's DB2 and remains attractive because of its self-contained nature. Although simple in principle, its details are tricky and little has been released to the public.

1. Gray and Reuter report ([6], p. 763) that Salzberg and Dimock have developed a method for RID reorganization. However, [9] is about transaction-based on-line reorganization and according to a personal communication, there is no other paper from them which considers dynamic RID reorganization.

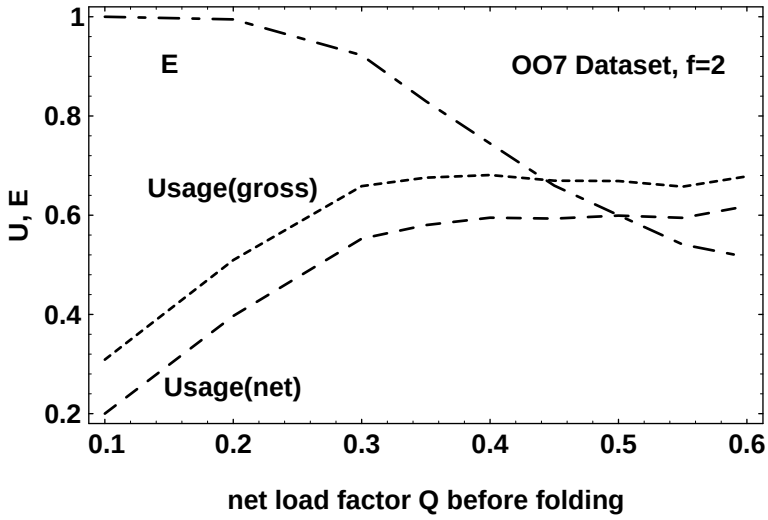


Figure 13: Gross and net utilization U and efficiency E of folding with $f = 2$

One particular problem is reclamation of empty space when such a RID-file becomes sparsely populated. Since Record Identifiers (RIDs, also called Tuple Identifiers, TIDs) are invariant by definition, pages can be deleted physically, but not logically. Therefore, there must be a mapping from “old“ to “new“ page numbers. If the self-contained nature is to be preserved, this is not to be achieved by a table but rather through some arithmetical “folding“ similar to hashing schemes. Page numbers are meant to collide creating merged pages.

The paper explains in detail an efficient division-folding method where f adjacent pages are merged into one. This process can be iterated should the load factor of the file continue to decrease. On the other hand, should the load start to increase again, the folding process can be reversed. The algorithms are also designed in a way that the reorganization can be done on-line and step-wise whenever the system is idle.

Like the RID-technique itself, the reorganization algorithm is simple in principle, but creates tricky special cases and has some not so nice features. One case is the fact that the folding process is best performed from left to right. This, however, does not free any pages for truncation until the very end. Another point is that empty slot lists inside the pages cannot be truncated and use up a fair amount of space in the merged pages.

Whether the last point is relevant depends very much on the actual type of data. As our empirical tests show, it is of concern when individual objects tend to be small on the average. On the other hand, few but large objects, as e.g. typical for multimedia databases, create insignificant slot lists but tend to block merging because the variance in page load after a random deletion process creates a surprisingly high number of incompatible pairs of pages. The resulting overflow is called a spill page and is chained onto the original location but rests at the end of the

file. These pages reduce the resulting utilization and in fact prevent load factors better than 67%.

These observations also suggest that folding should be combined with some type of object movement. Since displaced objects don't have reverse links, i.e. they don't know where they came from originally, this is not a trivial task and is left as an open research problem. Similarly, aspects of hiding the folding process within on-line transactions along the lines of [7, 9, 12] are not considered here and pose an interesting challenge.

Literature

[1] M.M. Astrahan et al. System R: Relational Approach to Database management, ACM TODS, 1:2 (June 1976) pp. 97-137

[2] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In Proc. 1993 ACM SIGMOD Conference, Washington D.C., May 1993, pp. 12-21

[3] H.Chou, D. DeWitt and A. Klug. Design and implementation of the Wisconsin Storage System, Software Practice and Experience 15, 10 (Oct. 1985)

[4] A. Eickler, C.A.Gerlhof, D. Kossmann. A Performance Evaluation of the OID Mapping Techniques, Proc. 21st VLDB, Zürich, Switzerland (1995) pp. 18-29

[5] A. Gamache and Nadjiba Sahraoui, Addressing Techniques Used in Database Object Managers O2 and ORION, SIGMOD Record 24:3 (Sept. 1995) pp. 50-55

[6] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques, Morgan Kaufmann Publishers, San Mateo, Calif., 1993

[7] E. Omiecinski, L. Lee, and P. Scheuermann. Concurrent File Reorganization for Record Clustering: A Performance Study, Eighth Int. Conf. on Data Engineering, Tempe, Ariz., Feb. 3-7, 1992, pp. 265-272

[8] Reda Khalifa Salama. Design of a Multi-Channel Operating System Based on UNIX, PhD-Thesis University of Kassel, Germany (1995)

[9] B. Salzberg and A. Dimock. Principles of Transaction-Based On-Line Reorganization, Proc. 18th VLDB Conf. Vancouver, B.C., 1992, pp. 511-520,

[10] J. Teuhola and L. Wegner: Minimal Space, Average Linear Time Duplicate Deletion, Comm. ACM 34:3 (March 1991) pp. 62-73

[11] L. Wegner, M. Paul, J. Thamm, and S. Thelemann: Pointer Swizzling in Non-Mapped Object Stores, Preprint 4/95 (August 1995) and Proc. Seventh Australasian Database Conference (ADC'96), Melbourne, Australia, 29-30 January 1996, Rodney Topor (Ed), Australian Computer Science Communications, Vol. 18 Number 2, pp. 11-20

[12] C. Zou and B. Salzberg. On-line Reorganization of Sparsely-populated B+-trees, Proc. 1996 SIGMOD, Montreal, Canada, SIGMOD Record 25:2, June 1996, pp. 115-125