

COSC-4411: Assignment #1

Due: 30 January 2003

1. Buffer Pool. *The Popularity Contest.*

A page that is pinned multiple times at some point (that is, its *pincount* was greater than one) is intuitively more popular than a page that was only pinned singly (that is, its *pincount* was equal to one). This is because several transactions were using the page at once.

However, none of the standard replacement policies—for instance, LRU nor Clock—account for this. In LRU, a page's popularity is measured by how recently it was last pinned (regardless of the *pincount*). Likewise in Clock, every unpinned page has an equal chance of replacement after having been passed over once.

Design a new replacement policy that is an adaptation of Clock which does take into account pages that were multiply pinned.

A solution is as follows. Keep for each frame a reference counter, ref_count. With the clock replacement strategy, a reference bit is usually kept. We are replacing the reference bit with ref_count.

Every time a page is pinned, the pin_count of the associated frame will be incremented, as usual. We modify the buffer manager to also increment ref_count of the associated frame too when the page is pinned.

Every time a page is pinned, the pin_count of the associated frame is decremented, as usual. However, we have not modified the buffer manager to also increment ref_count of the associated frame. The ref_count remains as is.

Thus ref_count is measuring the popularity of a page as laid out in the problem.

*Clock is modified as follows. Clock checks the current frame, and proceeds to the next if the frame's page is pinned. If the frame's page is unpinned (*pin_count* = 0), then it checks whether (*ref_count* = 0). If it is, this frame is used for replacement. Otherwise, clock decrements the frame's *ref_count*, and moves to the next frame.*

This allows more popular pages to remain in the buffer pool longer, because clock will pass over them more times before selecting them for replacement.

2. Buffer Pool. *I'm sorry. You're being replaced.*

A database is spread over two disks, **A** and **B**. An I/O read/write to **A** is ten times faster than an I/O to **B**. The same number of the database's pages reside on disk **A** as on **B**.

Should the replacement policy of the buffer manager be changed or modified from a standard replacement policy (such as LRU or Clock) because of this?

If so, how can the policy be changed to make the database system more efficient?

*To answer this rigorously and prove our solution is optimal would be quite hard. Also, we would need assumptions about access patterns off disks **A** and **B**. But we can outline a general solution.*

Our goal is to ensure that **B** pages have preference over **A** pages to be in the buffer pool. They are more expensive to fetch, therefore we need to increase the probability that a **B** page is already present when requested (at the expense of the probability that an **A** page is already present when requested).

A first idea is to give **B** pages total priority, which turns out to say always pick an **A** page as the replacement victim when possible. However, this could lead to bad performance. There generally would only be one unpinned **A** page in the buffer pool at any given time. And typically the ratio of unpinned to pinned pages in the buffer pool is high. As a policy, transactions unpin pages as soon as feasible. So few **A** pages are ever in the buffer pool. For any transaction that is **A**-page intensive (such as a range index retrieval of records all on **A**), this is a disaster.

The next idea then is to give **B** pages preference. How much? Intuitively, in the same ratio as cost; a **B** page should stay around roughly ten times as long as an **A** page. One way to accomplish this would be to modify the clock replacement strategy similar to what we did for Question 1. Use a `ref_count`; set it to 10 for a **B** page when unpinned, and to 1 for an **A** page when unpinned. This is better behaved. **B** pages are given preference, by **A**-page intensive transactions will still perform well since there is no limit placed on **A** pages being in the buffer pool.

This wasn't what I intended, but others considered strategies to migrate pages from disk **B** to **A**, and to keep disk **A** full. Indeed, this is a good idea generally. There are problems to implement this at the buffer manager, however. RIDs (record IDs) often are `page# + slot#` by design. Rolled into that `page#` is which logical device (disk) it is from. The buffer manager cannot reassign a disk-page's ID easily. Lots of bookkeeping would be effected. All indexes involving that page would have to be changed! Also, any special structure like physical sequential layout on disk would be lost in migration.

We could engineer a migration solution, but it would most likely involve the disk space manager and the file handler layers.

Final solution: Get rid of disk **B** and buy another like disk **A**. Indeed, this is not the type of problem real database system builders are going to pay attention to. They assume you'll use reasonable hardware.

3. Page Organization. Free the space!

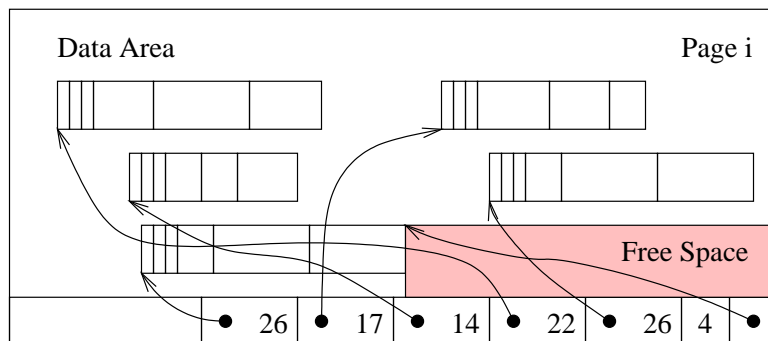


Figure 1: Page layout with variable length records.

Consider a page layout with variable length records and a slot directory as in Figure 1.

Say that we want to add a new record to page i , but there is not enough space for the new record in page i 's *free space*. However, there is enough space overall on the page for the new record. That is, if the records already on page i were rearranged ideally, then the new record would fit. Namely, we have to *pack* the records on page i : move them all so that they are sequentially adjacent starting from the beginning of the page. After this, all the empty space does belong to *free space*.

Write a pseudo-code algorithm for packing a page.

Caution! The records may not be in the same order physically on the page as their slot#'s. This is because we are supporting variable length records. Over time with inserts and deletes, things may become scrambled.

What is important is that we maintain a record's slot# on the page, since this is part of its address.

- *Copy the slot directory into auxiliary memory.*
- *Sort it by the offset field ascending (where on the page that record begins), dropping unused slots. Record in the sorted list slot# too.*
- *Set offset = 0 For $i = 0$ to $r - 1$ (where r is the number of active slots)*
 - *Look at entry i in our sorted slot directory.*
 - *Move the record R to offset on the page.*
 - *offset+ = $R.length$*
 - *Update the original slot directory at the correct slot# with the new offset.*

This way, we never run the danger of overwriting part of a record while moving. If we didn't sort the slot directory by offset but did the same procedure as above (which many proposed), records could get overwritten.

Another solution proposed is to allocate a new page, copy the records to the new page, and then move the new page to the old page's frame. Yes, this would work. But the inplace method is more elegant; it saves us from needing an extra page of space and the extra copying of the records.

4. **Search Keys and Indexes.** *An index by any other name...*

Consider the table **Student** with six attributes name, age, address, year, gpa, and major.

Does it make sense to have both **major + year** and **major + year + gpa** as tree indexes? (Recall, the order of the attributes of an index's search key is important.) Briefly, why or why not?

If both indexes are potentially useful for lots of the database's queries, but you can only keep one of them, which one would you keep? Why?

*Any query that can be answered (efficiently) by **major + year** can also be answered with the same degree of usage of the index by **major + year + gpa**.*

*Use of **major + year + gpa** might be slightly less efficient. Its fan-out in its index pages will be less since its search key is larger. So it may be deeper, requiring more I/O's. Likewise, there will be more data-entry pages (under alternative two or three) here. However, we expect the differences in performance to be slight.*

Were these differences in performance to matter to an important, expensive-to-evaluate, often asked query, maybe we would support `major + year` too. Otherwise, no.

So, if I could only keep one? If there are any anticipated queries that could use `major + year + gpa`, but not `major + year`, then clearly `major + year + gpa`.

If there are no anticipated queries that could use `major + year + gpa`, but not `major + year`, and lots that would benefit by `major + year`, then `major + year`. It is the smaller index and is less expensive to maintain.

5. **Search Keys and Indexes.** *An index by any other name...*

Consider the table **Student** with six attributes `name`, `age`, `address`, `year`, `gpa`, and `major`.

Say that the following indexes exist on **Student**:

- A. `name + gpa`
- B. `gpa + name`
- C. `major + year`
- D. `name + age + year`

Assume each index is good for both range and equality queries.

For each of the following queries, say which of the indexes possibly could be used to evaluate the query, and for each, briefly in what way the index could be used to advantage. State *all* that could apply. If none is applicable, say so. If more than one is applicable, which would most likely be the best choice and why? Or maybe none is.

- a. `select * from Student`
 where `gpa ≥ 8.0`;

B is the only candidate. Yes, A has the information `gpa` within. However, we cannot navigate into that index without a specific `name` value.

It is possible some query processors might consider the option of scanning A's data entry pages in a type of index-only plan, since this is likely many fewer pages than the data-record file itself.

- b. `select * from Student`
 where `name = 'George P. Burdell'`;

Now A and D are candidates. A is probably preferred since the index will involve fewer pages.

Again, B might be an option in a type of index-only plan, scanning the data entry pages.

- c. `select * from Student`
 where `age = 24`;

Nothing matches.

D might be an option in a type of index-only plan, scanning the data entry pages.

- d. `select * from Student`
 where `major = 'computer science' and age ≥ 19`;

C partially matches can be used. It can return all the 'computer science' records. Each will need to be checked for whether `age ≥ 19`. The index does not "evaluate" that predicate for us.

6. **I/O.** *I/O, I/O, it's off to work we go.*

Consider file **R** which has 10,000,000 records, and that we have a B+ tree index on this file with the search key as **R**'s primary key. Assume that the order (d) of the B+ tree is 60, and that the index pages are 80% full, on average.

- a. Assume that the B+ tree index above for **R** is of alternative *one*; that is, the leaf pages of the B+ tree contain the data records themselves (at 20 data records per page).

You want to find all records with search key $\geq x$ and search key $\leq y$. Say that 1,000 data records qualify. How many I/O's does it cost you to retrieve these?

Depth of the index to the first leaf page? We need to know the average fan-out: An index page can hold $2d$ index records, or 120 of them. They are on average 80% full, so 96 index records. This means a fan-out of 97, as the 96 keys can redirect us to 97 pages.

We need the index to get us to the right leaf page. Since this is alternative one, the leaf pages are the data-record pages. There are $1,000,000/20$ data-record pages, so 50,000 pages. (Okay, I didn't make clear whether a page could hold 20 records and then they were on average 80% full. If so, then 16 records per page on average. Or if what was meant was that, on average, a data-record page holds 20 records. Let's assume 20 records per page.)

So, $\lceil \log_{97} 50,000 \rceil + 1 = 4$. This includes the leaf page. So $\lceil \log_{97} 50,000 \rceil$ page I/O's to navigate the index pages from root to the appropriate lowest index page, and one more to grab the leaf page. (I messed this up talking in class and added one more!)

We are performing a range fetch. Thus, we use the index to locate the leaf page with the record that starts our range. Then we read leaf pages (data-record pages here) until we exceed the range. We estimate we are fetching 1,000 records. These could fit on $1,000/20$ leaf (data-record) pages, or 50. Okay, probably 51, since our first matching record might be on the middle of a page.

So 3 index pages + 1 leaf page (start of range) + 50 more, for a total of 54 I/O's.

- b. Assume that the B+ tree index above for **R** is of alternative *two* and is unclustered; that is, the leaf pages of the B+ tree contain data entries that are $\langle \text{key}, \text{rid} \rangle$ pairs and not the data records themselves. Say that 50 data entries fit per page, and say that your buffer pool is absurdly small with just five frames.

Again, you want to find all records with search key $\geq x$ and search key $\leq y$, and 1,000 data records qualify. How many I/O's does it cost you to retrieve these?

What is different than the last question? This index is unclustered and is of alternative two.

The index needs to index $1,000,000/50$ data-entry pages, so 200,000 pages. Depth of the index? $\lceil \log_{97} 200,000 \rceil + 1 = 4$.

Once we land on a leaf (data-entry) page, we need to read over $1,000/50 = 20$, pages. Well okay, 19 more leaf pages. Well okay, so the first match data-entry might be in the middle of the page, so 20 more leaf pages.

For each data entry, we need to fetch the data record. Each of the 1,000 data records is likely to be on a different page, since there are 500,000 pages of records. Furthermore, even for the rare cases there are two records we need on the same page, our buffer pool allocation (five frames) is so small, the likelihood the same page is still present when we

need it again is vanishingly small. So, it will cost us in worst-case—and here, average-case too—1,000 I/O's to fetch the data records, one I/O / page per record.

Grand total then is 3 index pages + 1 leaf (data-entry) page (start of range) + 20 more leaf (data-entry) pages + 1,000 data record pages, or 1,024 in all.

- c. Now assume that we have an index on **R**'s primary key instead that is an extendible hash index. Assume that the index is of alternative *two* and is unclustered. We want to find the record with search key value x . How many I/O's does it cost you to retrieve the record?

One page to read the right directory page, one page to read the bucket page containing the data entry (no overflow pages presumably for an extendible hash), and one page to get the data record. So 3 I/O's.