G+1 0 More Next Blog» lixincun@gmail.com Dashboard Sign Out

Some thoughts on your (and my) thoughts.

Just something whacky to get you thinking. At least it got me pulling apart my hair, and now, I'm half bald.

TUESDAY, NOVEMBER 24, 2009

An O(1) approach to the LFU page replacement algorithm

LFU: Least Frequently Used

The LFU algorithm evicts pages that have been **least frequently** used. If the cache is full, and one more page needs to be added, the page that has been used the **least** number of times is evicted from the cache.

How can we efficiently implement this algorithm? Each operation of the LRU algorithm can be implemented in worst case O(1) complexity. We can do the same for LFU as well.

For this, we maintain a doubly linked list where each node of the linked list looks like this:

```
struct CountNode
{
 CountNode *prev, *next;
 PageNode *root;
 int count;
};
```

Each node of this linked list points to the root of another doubly linked list which holds the actual pages. All the pages in this 2nd linked list have the same usage count. The node for this linked list looks like this:

```
struct PageNode
{
 PageNode *prev, *next;
 CountNode *parent;
 char pageData[4096];
};
```

About Me

Dhruv Matani

http://dhruvbird.com/

View my complete profile

Labels

algorithms	(19)
asynchronous I/O	(1)
attitude	(1)
biking	(1)
data structures	(10)
disgusting	(1)
duckduckgo	(2)
dyslexia	(1)
environment	(1)
fenwick tree	(1)
filesystems	(2)
firefox	(1)
food	(12)
fun	(7)
GRE	(1)
health	(6)
http	(1)
humour	(1)
idea	(7)
insightful	(25)
jabber	(1)
javascript	(6)
jquery	(2)
kerala	(1)
moments	(1)
mongrel2	(1)
movie	(1)

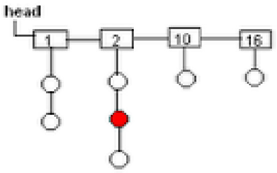

Figure-1: Configuration at some time

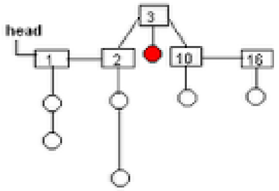

Figure-2: Configuration after the red node is accessed once

The PageNode's parent member points to that CountNode of which it is a child. So there is a direct pointer from every PageNode to the corresponding CountNode it falls under. These extra links are not shown in the diagram to keep it simple.

The CountNode linked list's count member holds the usage count of each element under that node. So each PageNode under a CountNode having a count value of 6 will have been accessed 6 times.

How to implement the various operations?

1. **Insetion:** To insert, we always add the new PageNode to the CountNode list with the count value of 1. This will always be the first node in the linked list pointed to by *head*. If the first CountNode pointed to by *head* is greater than 1, we just add a new CountNode with a count value of 1 and then perform the insertion. All this can be done in time **O(1)**.

2. **Deletion:** We will delete nodes, only if the cache is full, and this deletion will always happen from the CountNode list which has the least count. So, we just follow *head*, and delete a PageNode in such a CountNode. Additionally, if that CountNode happens to become empty because of this deletion, we also delete that CountNode. This too takes time **O(1)**.

3. **Access:** Access involves incrementing the count of a PageNode, i.e. moving the accessed PageNode from it's CountNode list to a CountNode list with a count value that is **1** greater than it's current count value. We maintain pointers to the CountNode in the PageNode. This makes deleting the PageNode(and possibly the CountNode to which it belonged) a constant time operation. After that is done, we can insert a new CountNode that has a count value of 1 more than the CountNode from which the accessed PageNode was removed. If such a CountNode does not exist, we can always create it first. All this can also be done in constant time, making the whole process **O(1)**.

mumbai (1)
node.js (5)
oop (2)
p2p (1)
patterns (4)
php (1)
poetry (2)
presenting (1)
programming (54)
python (4)
random (11)
rant (5)
recipe (33)
scaling (2)
security (1)
segment tree (1)
sqloptimization (10)
storage (3)
svd (2)
technical (64)
thecruftoftheworld (1)
tips (1)
tool (2)
tree (3)
tutorial (18)
vacation (2)
vanilla (1)
weather (1)
xmpp (1)
yoga (2)

Blog Archive

► 2015 (7)
► 2014 (7)
► 2013 (10)
► 2012 (16)
► 2011 (28)
► 2010 (54)
▼ 2009 (21)
▼ 11/22 - 11/29 (1)

A lot of people have asked me how they can get to a PageNode in the first place. The answer to that question is that we maintain a hash table which hashes an object(page address, integer, etc...) such that accessing it (getting to the PageNode that that object corresponds to) can be accomplished in O(1) time.

This implementation takes advantage of the fact that:

1. When we add a new object to the cache, it is always going to start off with a use count of 1, and
2. Whenever we access an object, it's use count will go up by exactly 1.

I haven't found any resource that says that LFU can be implemented in time O(1). If you (reader) find any, please do let me know.

Posted by Dhruv Matani at 8:12 AM
Labels: programming, tutorial

6 comments:

Vijay said...

Will this algorithm take O(n) time in the following scenario?

Assume that all the n pages have use count C. Then there is a big list of page nodes attached to one count node representing C. Now if we want to access a random page node in this big list of page nodes, we may have to traverse that big list of page nodes, which in worst case is O(n)

Tuesday, November 24, 2009 9:05:00 PM

dhruv said...

The page accesses happen through a separate hash table so that you can lookup any page within this data structure in amortized constant time. We never use this data structure to do page lookups.

This is the same way that lookup happens in an LRU cache replacement algorithm.

Hence, it will still be O(1).

Tuesday, November 24, 2009 10:41:00 PM

Gunjan said...

I guess one way to implement this hash would be to use an array, where a particular index stores the reference to the PageNode with that page number. Thus a constant time lookup.

Tuesday, November 24, 2009 10:56:00 PM

An O(1) approach to the LFU page replacement algor...

- ▶ 11/15 - 11/22 (2)
- ▶ 10/25 - 11/01 (1)
- ▶ 10/11 - 10/18 (1)
- ▶ 10/04 - 10/11 (1)
- ▶ 07/26 - 08/02 (2)
- ▶ 07/19 - 07/26 (1)
- ▶ 07/12 - 07/19 (4)
- ▶ 06/28 - 07/05 (2)
- ▶ 05/10 - 05/17 (1)
- ▶ 04/12 - 04/19 (2)
- ▶ 02/01 - 02/08 (1)
- ▶ 01/11 - 01/18 (1)
- ▶ 01/04 - 01/11 (1)

- ▶ 2008 (25)
- ▶ 2007 (24)
- ▶ 2006 (11)
- ▶ 2005 (14)
- ▶ 2003 (3)

Blogs/Folks I like

- Anirban Mitra
- Kamala Yazhini
- Zed Shaw
- Pieces of Rakesh
- Apurva's Blog
- Pascal's Blog
- Sandy's Blog
- Ramya's Blog

dhruv said...

Yes, and since the cache size is fixed, we can actually get O(1) performance in most of the cases, if we choose a good hash. I mean there will be a negligible number of collisions if we choose a good hashing function.

Tuesday, November 24, 2009 11:06:00 PM

Ved said...

http://ved-antani.com/2011/12/javas-linkedhashmap-demystified-sort-of/

Sunday, January 08, 2012 1:12:00 AM

hc David said...

Have a look at this paper. http://dhruvbird.com/lfu.pdf

Thursday, January 23, 2014 10:40:00 PM

Post a Comment

Links to this post

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)