CS 186 Fall 2024 | Introduction to Database Systems Alvin Cheung | DIS 4

1 Buffer Management

(a) Fill in the following tables for the given buffer replacement policies. You have 4 buffer pages, with the access pattern **A B C D A F A D G D G E D F**

Least Recently Used (LRU)

A				*		*							F	Hit Rate
	B				F						E			$\frac{6}{14}$
		C						G		*				
			D				*		*			*		

Most Recently Used (MRU)

A				*	F	A								Hit Rate
	B													$\frac{2}{14}$
		C												
			D				*	G	D	G	E	D	F	

Clock ("second chance LRU")

A				*	F							D		Hit Rate
	B					A							F	$\frac{4}{14}$
		C						G		*				
			D				*		*		E			

Red frame outline is where the clock hand points at the end of the access
Yellow frame outline is where the clock hand considered ejecting a page, but the bit was one
Red box means the second chance bit is zero
Green box means the second chance bit is one
* means a hit

(b) Fill in the following tables for the given buffer replacement policies. You have 4 buffer pages, with the access pattern **A, B, C, D, A, F (remains pinned), D, G, D, unpin F, G, E, D, F**
Remember that unpinning does not contribute to the hit count!

Least Recently Used w/ Pinning

A				*								E			6/13
	B				F	x	x	x	x	x				*	
		C						G			*				
			D				*		*				*		

Most Recently Used w/ Pinning

A				*	F	x	x	x	x	x	G	E			3/13
	B														
		C													
			D				*	G	D				*	F	

* = hit
X = pinned

Remember that unpinning doesn't contribute to the hit count

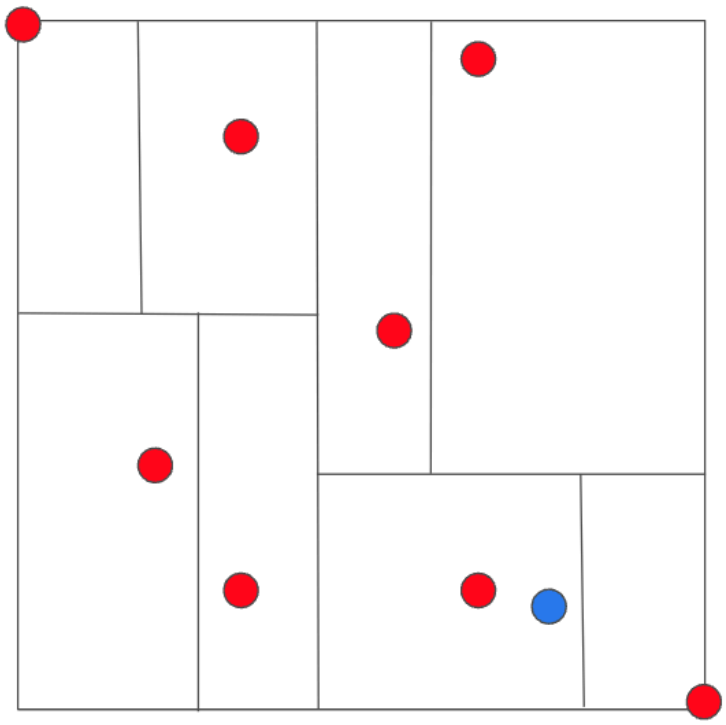
(c) Is MRU ever better than LRU?
Yes; MRU prevents sequential flooding during sequential scans

(d) Why would we use a clock replacement policy over LRU?
Efficiency (approximation of LRU; don't need to maintain entire ordering)

(e) Why would it be useful for a database management system to implement its own buffer replacement policy? Why shouldn't we just rely on the operating system?
The database management system knows its data access patterns, which allows it to optimize its buffer replacement policy for each case

2 Nearest Neighbors with K-d Trees

Consider the image below, which shows two dimensional database points (red) partitioned by constructing a k-d tree. We wish to use our k-d tree to find the nearest neighbor to the query point (blue). Suppose that we use the Euclidean distance as our metric.



(a) How many distance calculations between our query point and database points would be needed in a brute force search?

We need to do 8 distance computations as that is how many database points there are. We need to check the distance to each one.

(b) How many distance calculations between our query point and database points would be needed if we use a k-d tree instead?

We only need to do 2 distance computations. First we check the distance to the database point in the same box as the query point. However, since the distance to the box to the right is less than the distance to that database point, we must also check the box to the right to make sure that we have found the correct nearest neighbor. When we check the box, we do another distance computation with the point in that box. Since no other boxes can contain a nearer neighbor, we do not need to do any more comparisons.

3 Curse of Dimensionality (Optional)

In this question we will explore why nearest neighbor search using k-d trees degrades from logarithmic time to linear time complexity in higher dimensions. This provides motivation for approximate nearest neighbor indexes, which sacrifice accuracy in exchange for a reasonable runtime.

(a) Suppose we have a database of 1,000 d-dimensional random data points in [0, 1]^d. Given a d-dimensional query data point, we want to find the closest data point in the database. In a 1-dimensional space, starting from our query point, how much of the interval [0, 1] do we expect to search?
We expect to search an interval of length 1/1000 = 0.001.

(b) Repeat the above but now in a 2-dimensional space. In each of the 2 dimensions, how much of the interval [0, 1] do we expect to search? What about in a 3-dimensional space?
In two dimensions, we need to search each dimension so that the total area of our search space is 0.001. Thus we need to search a length of (0.001)^(1/2) = 0.031 in each dimension. In three dimensions, we need to search so that the volume of our search space is 0.001. Thus we need to search a length of (0.001)^(1/3) = 0.1 in each dimension.

(c) What about in a 128-dimensional space?
(0.001)^(1/128) = 0.947. We need to search almost the entire length in each dimension to find the nearest neighbor!

(d) Why is this problematic for k-d trees and other tree indexes which rely on spatial partitioning?
Since we expect to search almost the entire length in each dimension, the nearest neighbor search using a k-d tree will need to check nearly all the branches to make sure it has found the real nearest neighbor.