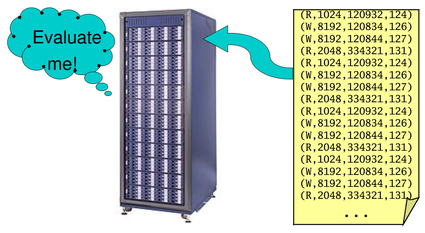
Distributing Jump Distances in a Synthetic Disk Array Workload

Zachary Kurmas and Jeremy Zito, Grand Valley State University

Background

Workloads

Workload traces (lists of individual I/O requests) are often used to evaluate storage systems



Where do these traces come from?

Real vs. Synthetic Traces

Real: List of I/O requests made by an application in a production environment.

- Large
- Inflexible
- Hard to find due to security issues
- Perfectly accurate

Synthetic: Generated randomly to maintain high level properties of a workload.

- Compact
- Easily modified
- Contains no specific data
- Rarely accurate

Increasing synthetic workload accuracy

Jump Distance

An accurate synthetic workload must cause the disk array to behave as if it were running some real workload (called the *target* workload). To do so, it must share certain key properties with the target workload. We have found that the distribution of *jump distance* is one property that must be shared.

Problem

Jump distance is an approximation of the distance the disk head moves between disk accesses, and is defined by (***startSector[x] – startSector[x-1]***). Jump distance affects a storage system's behavior because the physical movement of the disk heads is often the primary bottleneck in a storage system. To create an accurate synthetic trace, we must maintain not only a correct distribution of jump distances, but also a correct distribution of sectors accessed. Generating a synthetic list of sectors accessed that maintains both distributions reduces to the NP-complete Hamiltonian Path problem.

Our Solution

We generate a synthetic disk access pattern (i.e., the list of sectors accessed) by solving the Hamiltonian Path problem that is defined by the target workload's distributions of starting sectors and jump distances. We first create a directed graph that contains (1) a vertex for each sector accessed, and (2) an edge between each pair of sectors that correspond to some jump distance in the target workload. We then search for a Hamiltonian path using a depth-first search. Each Hamiltonian path corresponds to an access pattern that maintains the target workload's distribution of sectors accessed and jump distance.

Algorithm

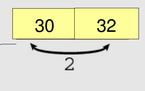
Building the Graph

We first analyze the target workload and measure how often each sector is accessed and how often each jump distance occurs.

... 30 ...
... 32 ...
... 42 ...
... 2 ...
... 32 ...
... 30 ...
... 4 ...

For example, the workload segment to the left has jump distances of 2, 10, -40, 30, -2, and -26.

We then build an adjacency matrix based on the sectors accessed and the observed jump distances. Because the target workload accesses sectors 30 and 32, we add an edge between *any* two sectors with a jump distance of 2 --- including sectors 2 and 4 from the example to the left.



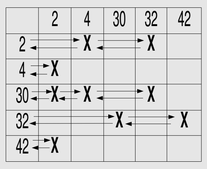
	2	4	30	32	42
2		X			
4					
30				X	
32					
42					

Notice that we also added an edge between 30 and 32 because there is a jump of 2 between them.

Adding the remaining edges produces this adjacency matrix:

	2	4	30	32	42
2		X		X	
4	X				
30	X	X		X	
32			X		X
42	X				

The "X"s in each row represent those sectors that are allowed to follow the sector listed at the head of the row. Each row is placed in a linked list.



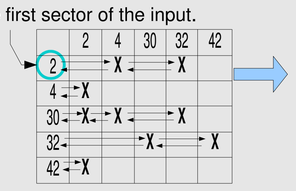
Main Loop

We then search the graph for a path that will maintain the sector and jump distance distributions of the target workload.

The Search

Using the graph of possible paths, we begin the depth-first search.

- We must first choose a starting sector. For simplicity we use the first sector of the input.
- Possible paths from 2 are 4 and 32. We choose 4 ---the first on the list.



We set sector 4 as the current sector. The search continues by looking for the first sector in the list for row 4 . In this case, 2 is the only possible path.

Synthetic workload:

2	4	2	...

Now the current sector is once again 2. This is a problem because the synthetic workload must maintain the target workload's distribution of sectors. Because sector 2 is used only once in the original workload, we remove it from the linked lists so that it can no longer be used.

Hiding a sector is simply a matter of removing its links.

For example, when sector 2 is hidden after its first use, it is bypassed in the linked lists.

A similar method of hiding is done if the maximum number of jump distances is exceeded (because the distribution of these must be the same as the original workload).

It may be the case that the search encounters a sector that has no possible paths leading from it. For example, sector 4 no longer has a path to 2 because 2 has been hidden. This leaves no possible paths from sector 4. We then backtrack and take the next available route.

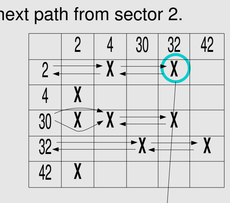
Backtracking

In our example, sector 4 is a dead end:

Synthetic workload:

2	4	... X

We must then backtrack and chose the next path from sector 2.



The next path is 32. After adding the new path, the search continues:

Synthetic workload:

2	32	...

While backtracking, we look for possible hidden nodes that become available. Because backtracking "adds back" sectors and jump distances, they may become available again. In our example, if there had only been one sector 4, it would have been hidden after it was added:

Synthetic workload:

2	4	...

However, when we backtracked and chose the alternate path to sector 32, sector 4 is "added back" and can then be unhidden because it no longer appears in the synthetic workload.

Synthetic workload:

2	32	...

* Note that as the synthetic workload is built, we may have to backtrack many times.

Finish Up

Because the depth-first search may not finish in a reasonable amount of time, we stop it when it fails to make progress for a set number of iterations. We then use approximation techniques to distribute the remaining sectors.

Finish Randomly

"Finish randomly" is just what it sounds like: the random placement of those sectors "left over" when the search is terminated because it is not making any progress. This random placement of sectors could have an adverse affect on jump distance distribution if the number of "leftover" sectors is large.

Synthetic workload:

546	1247	443	780	2140	2543	1247	743	2477	8345	346

Leftovers:

226	3460	1683	360

Final Synthetic Workload:

546	1247	1683	443	780	2140	3460	2543	226	1247	743	2477	360	8345	346

Results

The important question is: ***How far does the algorithm get***?

Because the Hamiltonian Path problem is NP-complete, we don't expect to find a complete path in a reasonable amount of time. The length of the path found before the search stalls is largely determined by the order in which sectors are searched. When we build our adjacency matrix, ***what happens if we change the order in which sectors are searched***?

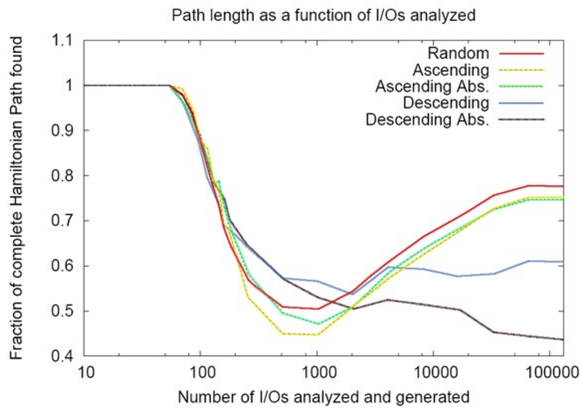
To determine the best way to order our adjacency matrix, we add sectors in one of the following ways:

- **Ascending:** Examine the I/O requests in order from smallest to largest starting sector.
- **Ascending absolute value:** Examine the I/O requests closest to the current sector first.
- **Descending:** Examine the I/O requests in order from largest to smallest starting sector.
- **Descending absolute value:** Examine the I/O requests farthest away first.
- **Random:** Assign a random order.

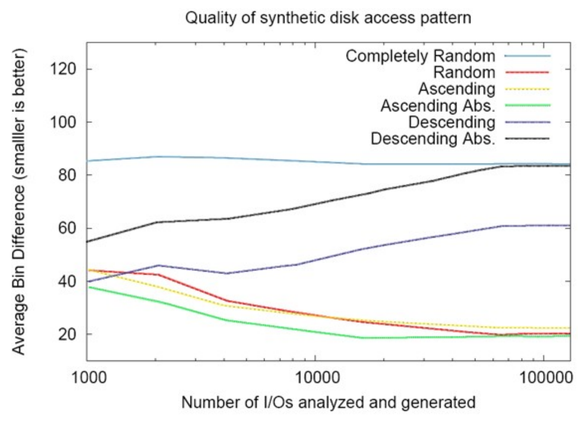
For Example, a random ordering may create the following adjacency matrix:

	2	30	42	32	4
2				X	X
30	X			X	
42					
32		X	X		X
4	X				

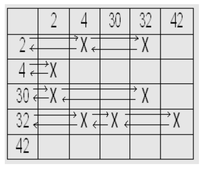
The graphs to the right analyze a synthetic workload, as generated by different sort orders. The synthetic list is based on a "real" workload from an e-mail application.

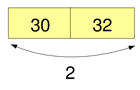


The longest partial Hamiltonian path is created when using a random ordering of sectors. The next best sort is ascending. A random ordering produces a different solution every time the algorithm is run. In contrast, the other orderings produce the same results. Also, notice that a complete path is found only for workloads with less than 100 I/Os. However, most workloads are very large, containing thousands and thousands of I/Os.



One metric used to determine the quality of a synthetic workload is "average bin difference." This is simply the difference between the number of jump distances used in the "real" workload and those used in the synthetic workload. For example, a jump distance of 2 may be found 20 times in the "real" workload, but only 12 times in the synthetic workload. This difference is calculated for every jump distance, and the average difference is then calculated. Once again, we see that the random and ascending search orders produce more accurate access patterns.




30
32
2