Flexible Cache for Database Management Systems

Radim Bača and David Bednář

Department of Computer Science, VŠB – Technical University of Ostrava
17. listopadu 15, 708 33 Ostrava-Poruba, Czech Republic
{radim.baca,david.bednar}@vsb.cz

Abstract. Cache of a persistent data structure represents an important part, which significantly influence its efficiency. Cache consists from an array of main memory blocks (caled cache nodes) with a constant size. Cache nodes buffer the data structure nodes hence they can be accessed quickly. However, data structure nodes do not usually fully utilize the whole main memory block. Therefore, the constant cache node size the waste of the main memory. In this article, we propose the solution where the cache consists from the cache nodes with a different size. Cache is split into several cache rows. The data structure nodes with the similar size are stored in one cache row. Such a solution has two problems: (1) that the some cache row has to replace the nodes too often during the querying (cache row contention), and (2) that the data structure nodes has to be moved between the cache rows during the insert and the delete operation. In our experimental section we show that the effect of the cache row contention is minimized if we set the cache row size according to the data structure node distribution. We also discuss a solution of the node cache moves problem. **Key words:** Cache utilization, flexible cache

1 Introduction

The cache of a persistent data structure represents an important part, which significantly influence its efficiency. It consists from an array of main memory blocks called *cache nodes* with a constant size. Cache nodes buffer the *data structure nodes* hence they can be accessed quickly. Note the difference between the cache node and the data structure node which is a logical node of a data structure and it does not have to be loaded in the cache. Major research effort focus on a cache *replacement algorithm* [3, 1, 2], which selects the node that should be removed from the cache array to create free space in the cache for new data structure nodes.

The cache is usually shared by many data structures which can have a different node size in a main memory. Another issue is that data structure node usually does not utilize the whole node. Therefore, the constant cache node size is waste of the main memory since the size of the cache node has to be equal to a largest data structure node in the database system. Similar situation occurs when we use compressed data structures [4] which compress the node when it is stored on a secondary storage, but the node is decompressed in the main memory. Usage of compressed data structures highlight the problem of variable size of a data structure nodes, since two nodes can have a significantly different main memory size even though their compressed size is similar.

V. Snášel, J. Pokorný, K. Richta (Eds.): Dateso 2011, pp. 146–152, ISBN 978-80-248-2391-1.

In this article we propose the solution called flexible cache containing cache nodes with a different size. Flexible cache is split into several cache rows. Nodes with the similar size are stored in one cache row. It can happen that we have to replace nodes from one cache row more often that in other cache rows. We call this problem *cache row contention*. We size the cache rows according to the node distribution in order to minimize the impact of this problem and to utilize the advantages of the flexible cache. Another problem called the *node cache moves* occurs during the insert and the delete operation. As we insert items into a data structure node its utilization grows. When it becomes too large then we have to move it into another cache row with the bigger cache nodes.

In Section 2, we describe different cache strategies. Section 3 introduces basic ideas of our flexible cache strategy and solutions of its problems. In Section 4, we describe our experimental results and in Section 5 we depict possible future improvements of the flexible cache.

2 Cache of Persistent Data Structures

The databases systems retain the data structure pages in cache nodes for a period of time after they have been read in from the disk and accessed by a particular application. The main purpose is to keep popular nodes in memory and reduce the disk I/O.

There is a number of different cache strategies which are usually used in the persistent data structures. Data is stored in *blocks* on a secondary storage. Block has a constant size varying typically from 2kB to 16kB. Cache contains a set of cache nodes which can be used in order to accommodate data from the block. When a data structure needs a data from some block then it is first loaded into some selected cache node. Selected cache node usually corresponds to a different data structure node, therefore, we have to write the node into a secondary storage first (if the node contains any changes) before we load the new node there. Selection of an appropriate cache node is usually called the *replacement algorithm*. Basically, the major issue is to find a cache node which will not be needed for a longest period of time. Since this kind of prediction is usually not available, there is number of different algorithms commonly used:

- RR (Random replacement) - this algorithm select the node randomly when necessary. This algorithm does not require keeping any information about the access history of nodes.
- LRU (Least Recently Used) - there is a time-stamp assigned to every node. We actualize the time-stamp each time we access the node. This selection algorithm simply select the node with lower time-stamp. This is a basic algorithm with many different variants.
- LFU (Least Frequency Used) - algorithm counts number of node usages. Nodes used least often are discarded first. If the frequency of usage of each node is the same, then nodes are expired by the LRU algorithm.

There is a number of replacement algorithms which usually combines the LRU and LFU [3, 1, 2]. LRU-K algorithm [3] remembers k recent accesses to the node and selects the node considering all these accesses. It is a generalization of the LRU algorithm which is considered as a LRU-1. In the next, we discuss some standard methods.

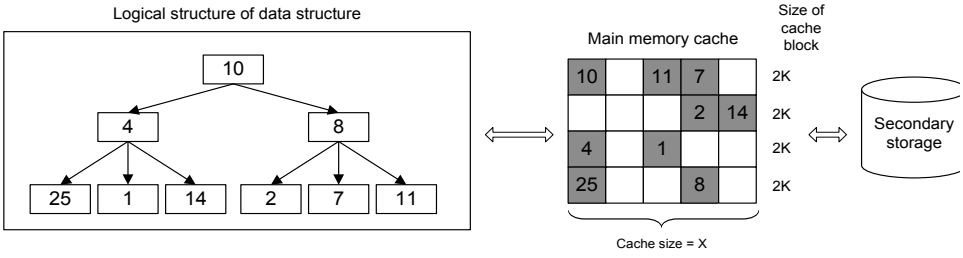

Fig. 1. Cache between physical storage and logical structure - cache size is setup on some constant value of blocks (X). Each block has the same own size, in this case 2K

2.1 LRU-K Page Replacement Algorithm

The basic idea of LRU-K ($k \geq 2$) method is to keep track of the times of the last k references to popular nodes, using the information to statistically estimate the interarrival times of references on a node by node basis. The LRU-K algorithm has many salient properties. It differentiates between the nodes with different levels of reference frequency, it detects locality of reference within query executions across multiple queries in the same transaction or in a multiuser environment. It is self-reliant (not need any external hints) and fairly simple. LRU-K keeps in the memory only those nodes that seem to have an interarrival time to justify their residence. That residence can be nodes with the shortest access interarrival time or equivalently greatest probability of reference, e.g. LRU-1 ($k = 1$) algorithm keeps in the memory only those nodes that seem to have the shortest interarrival time. The algorithm is based on the two data structures:

- K most recent references to nodes n from the history point of view - $HIST(n, x)$, where x represents the time of the last (the second to the last and so on) reference ($x \in 1, ...k$).
- Denotes the time of the most recent reference to node n - $LAST(n)$.

The LRU-2 algorithm provides essentially optimal buffering behavior based on the information given. It has significant performance advantages over conventional algorithm like LRU-1, because discriminate better between frequently referenced and infrequently referenced nodes.

3 Flexible Cache

Here we discuss the ideas behind the flexible cache which allows us to utilize the cache more efficiently.

The nodes of data structures can have a different node utilization, however, database management systems use cache nodes with constant size which is equal to the block size. This leads to a situation that main memory is wasted. In the flexible cache we solve this problem by creating the cache nodes with a different size. This reorganization

allows to us store more cache nodes in the main memory which consequently reduces the number of disk accesses which are very time consuming.

Cache is split into several *cache rows* r, where the cache nodes belonging to the same cache row has the equal size *nodesize(r)*. Each cache row r has its own size *count(r)* which represents the number of cache nodes belonging to r. There is a set of cache rows r_1, ..., r_m, where *nodesize(r_i)* > *nodesize(r_{i+1})*. Every data structure node n has its *realsize(n)* which specifies what is the minimal size of main memory where the node in the current state can be stored. The *realsize(n)* is mainly driven by the node utilization.

Data structure node n belongs to the the row r_i:

- If *realsize(n)* < *nodesize(r_i)* and *realsize(n)* > *nodesize(r_{i-1})*, or,
- if i = 1 and *realsize(n)* < *nodesize(r_i)*.

We have to slightly adapt the replacement algorithm for our flexible cache. The node n which should be loaded into the cache belongs to a cache row r_n, therefore, the replacement algorithm search for a appropriate cache node only in row r_n.

Figure 2 shows us the idea on the example. We have the cache with four cache rows. The first row r has cache nodes with *nodesize(r)* equal to the size of a biggest data structure node. It is for the data structure nodes which are almost fully utilized. Other cache rows store smaller data structure nodes, where the smaller node size can be caused by many different parameters (node utilization, bad compression ratio in a case of compressed data structures, data structure with smaller nodes).

Flexible cache has two problems: (1) we could read more often from one cache row then from the other one (cache row contention), and (2) we may be forced to move the data structure node from one cache row to another during the inserts. Both problems has a solution which minimize their influence. We introduces solutions for both problems is Section 3.1 and in Section 5.

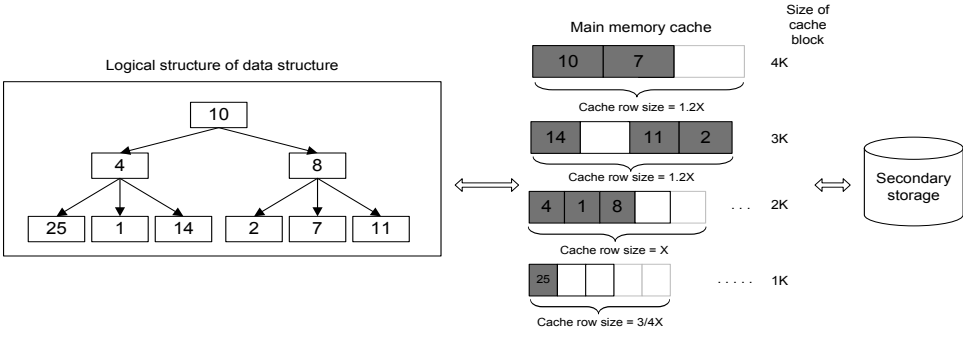

Fig. 2. Cache blocks with different size - cache is split into row with different size of blocks in every row - e.g. 4K to 1K. In the first cache row are stored nodes with the size 4K. Moreover, the rows have no constant and same size itself. The size of each row depends on percentage representation nodes with that size in the persistent structure.

3.1 Cache Row Contention and Node Cache Moves

If the *count(r)* some row r is too small and we replace nodes in this row too often (when compared to other cache rows) then we talk about the cache row contention. The problem can be minimized if we collect the statistics about the node distribution and cache rows are sized according to this distribution. As we will see in experiments, row contention was minimized and the flexible cache performed usually better then the regular cache.

The second problem could be minimized if we have a low priority thread in our database system which moves the data structure nodes from one cache row into the another one when the cache node becomes almost full. Each cache row r_i has a *threshold(r_i)* which is a value slightly lower than *nodesize(r_i)*. We keep the *move list* of nodes and the data structure node n is added to the list when its size exceed the specified *threshold(r_i)*. Thread then move the nodes from the list from one cache row into the another.

4 Experimental results

In our experiments[3], we use two-dimensional real data collection called TIGER (Topologically Integrated Geographic Encoding and Referencing system) containing about 5.5 millions records.

During tests we measure query time (range query randomly created), insert time (time of the creation the data structure) and Disk Access Cost (DAC). The flexible cache solution is compared with the common cache. The code is implemented in C++. Persistent data structure is in our experiments represented by R*-tree and compressed R*-tree data structure. The compressed R*-tree data structure is used because of the fact that *realsize(n)* $<$ *nodesize(n)* occurs more often than for R*-tree data structure. We use queries with 1%, 5% and 10% selectivity - meaning random queries which return x% of records from the data structure. We do our experiments with the two different cache sizes - 100 and 500 nodes. All R*-tree data structures used in our experiments are shown in Table 1.

From table 1 we can see creation time of a data structure with common and flexible cache. The creation time of a data structure with the flexible cache takes approximately 3% longer, than the creation of a data structure using the common cache. We can also see that an R*-tree data structure are created generally faster than a compressed R*-tree data structure (using the same data).

Tables 2, 3, and 4 show query processing results for the data structures with 5 millions records. The usage of flexible cache implementation decreases amount of disk accesses and time of query processing in most cases. In some cases the mentioned cache row contention negatively influence the query processing, but generally we can say, that total amount of DAC is reduced.

[3] The experiments were executed on an AMD Opteron(tm) Processor 865 1.80 GHz, 32 GB RAM, 64-bit Operating System Windows Server® Daracenter 2007 with Service Pack 2

TIGER data - Compressed R*-tree						
					creation time [s]	
tree (mil)	height	leaf nodes	inner nodes	size[MB]	flexible cache	common cache
1	3	4442	435	9.52	659.28	625.49
2	3	8891	759	18.8	1321.40	1319.83
5	4	22029	2125	47.1	3507.48	3486.38

TIGER data - R*-tree						
					creation time [s]	
tree (mil)	height	leaf nodes	inner nodes	size[MB]	flexible cache	common cache
1	5	8935	1294	19.9	231.63	225.42
2	5	17914	2015	38.9	482.40	466.15
5	5	44448	5386	97.3	1256.27	1193.23

Table 1. test collections - cache size used - 100 nodes

tree (mil)		R*-tree					
		flexible cache			common cache		
		1%	5%	10%	1%	5%	10%
5	DAC[kB]	58022	216470	370042	58176	211888	370764
	time[s]	3.08	14.08	24.42	3.19	12.50	28.06

Table 2. DAC and time for R*-trees - cache 100

tree (mil)		Compressed R*-tree					
		flexible cache			common cache		
		1%	5%	10%	1%	5%	10%
5	DAC[kB]	30890	112862	195690	33192	113608	198218
	time[s]	23.78	83.39	132.99	33.98	110.05	172.83

Table 3. DAC and time for compressed R*-trees - cache 100

As expected the insert (update or delete) operation is worse than the original solution because of the *cache move*. Inefficiency during *cache move* is from 2% to 5% mainly depending on a number of indexed labels in a data structure. The advantage of the flexible cache is particularly evident in a case of the compressed R*-tree. The flexible cache has aproximately by 30% better the query processing time in our tests in a case of the compressed R*-tree.

5 Future Work

In this article we presented a concept of the flexible cache which reduces number of cache node replacements. It also brings new chalenges such as cache row contention which can occur if some cache row is used more often then the others. We minimize

tree (mil)		Compressed R*-tree (compressed ratio 2)					
		flexible cache			common cache		
		1%	5%	10%	1%	5%	10%
5	DAC[kB]	28982	106486	183726	30054	107016	188298
	time[s]	18.56	85.17	150.22	29.90	119.62	192.92

Table 4. DAC and time for compressed R*-trees - cache 500

the cache row contention using the statistics about the node distribution, however, this approach does not have to work when we have many data structured where one is used more then the others. In the future work we would like to focus on this problem and extend our approach that cache row size is set automatically according to the workload. Therefore, the cache row size will be set according to the cache row usage and not according to the node distribution.

6 Conclusion

In this article we evaluate enhanced cache of the persistent data structure. The proper and right manipulation with the cache is very important for well results from our applications. There is a big performance lost when the cache is inefficient. In the section 3 we descibed our new investigation and solution to have good cache conscious data structure. Then in the serction 4 we confirmed our theroretical thoughts compared to practical tests. We could see that performance was increased and DAC were reduced. Because of some case was no better for new solution, we have open area for new and additional investigation to propose our solution more general.

References

1. T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450. Morgan Kaufmann Publishers Inc., 1994.
2. N. Megiddo and D. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130. USENIX Association, 2003.
3. E. O'neil, P. O'neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2):297–306, 1993.
4. J. Walder, M. Krátký, and R. Bača. Benchmarking Coding Algorithms for the R-tree Compression. In *Proceedings of the 9th Annual International Workshop on Databases, Texts, Specifications and Objects, DATESO*, pages 32–43, 2009.