Index Maintenance Strategy and Cost Model for Extended Cluster Pruning

Anders Munck Højsgaard, Björn Þór Jónsson, and Philippe Bonnet

IT University of Copenhagen, Copenhagen, Denmark
{amuh|bjth|phbo}@itu.dk

Abstract. With today's dynamic multimedia collections, maintenance of high-dimensional indexes is an important, yet understudied topic. Extended Cluster Pruning (eCP) is a highly-scalable approximate indexing approach based on clustering, that is targeted at stable performance in a disk-based scenario. In this work, we propose an index maintenance strategy for the eCP index, which utilizes the tree structure of the index and its approximate nature. We then develop a cost model for the strategy and evaluate its cost using a simulation model.

Keywords: High-dimensional indexing · Index maintenance · eCP

1 Introduction

In recent years, the scale and availability of multimedia collections has grown rapidly, spurring interest in scalable high-dimensional indexing methods. In some cases, including copyright protection and multimedia analytics applications, these media collections can be quite dynamic, with the most recently added material of particular interest. It is thus of interest to propose and study methods for dynamic index maintenance [7].

Overall, high-dimensional indexing structures fall into one of three main categories: tree-based, quantization-based, and hashing-based [7]. As many tree-based indexes inherit properties of their lower-dimensional counterparts, index maintenance has been discussed in some early works based on the seminal R-trees and kd-trees, as well as some more recent works [3,11]. In particular, dynamic index maintenance with transactional properties has been proposed for the NV-tree, the most scalable tree-based indexing structure [6,8]. Dynamic maintenance of index structures in the other categories, however, remains understudied.

Quantization-based methods have shown significant promise for scalability [1,5,10]; exploring strategies for dynamic maintenance is therefore of interest. In this paper, we consider dynamic maintenance of the Extended Cluster Pruning (eCP) indexing strategy [2,4]. Unlike many other quantization-based methods, eCP focuses on disk-based scalability scenarios by targeting a balanced cluster size and using an approximate hierarchical index to facilitate access to clusters.

In this paper, we propose a new strategy for maintaining eCP clusters that relies on the approximate nature of the hierarchical index. The strategy maintains the balanced distribution of cluster sizes while also allowing gradual reorganization of the high-dimensional space. We describe this new strategy and show, using a cost model-based simulator, that the strategy is efficient.

The remainder of the paper is organized as follows. In Section 2 we review the eCP high-dimensional indexing strategy. In Section 3 we outline the index maintenance strategy and its cost model. We briefly outline and analyze the performance of the strategy in Section 4, and give concluding remarks in Section 5.

2 Extended Cluster Pruning

Extended Cluster Pruning (eCP) is an approximate clustering-based high-dimensional search index. This index takes a dataset D of n vectors, where each vector consists of S_v bytes (including an identifier), and forms a set of clusters by randomly selecting a set of l cluster leaders. All vectors in D are then assigned to the closest leader; this process is essentially a single round of the k-means algorithm. When the collection is queried with a query vector q, the query vector is compared to the l cluster leaders to find the nearest leader l'. Then q is compared to all the feature vectors in the cluster of l' to find the (approximate) k nearest neighbors. Additionally, eCP has a parameter b used to expand the cluster search process, such that a search of the index returns the b clusters nearest to q, from which the k nearest neighbors then are found.

The motivation of eCP is to perform well in disk-based scenarios, where data only partially fits in memory and must thus often be read from disk. The overall goal of eCP is that each cluster read should typically result only in a single disk read, and three main techniques are used to achieve this goal.

First, the eCP targets a well balanced distribution of data across clusters by only performing a single round of assignments to cluster leaders. Experimental results with real datasets have shown that using a full k-means algorithm results in a highly skewed cluster size distribution, which in turn results in sub-optimal performance [4,9].

Secondly, eCP tries to have most clusters fit within the size of a single disk block read S_{io}, which has a default value of 128 KB for Unix systems.[1] In order to find this "optimal cluster size" in which we can fit the most possible feature vectors within the size of one S_{io}, one must calculate the following:

$$S_c = \lfloor S_{io}/S_v \rfloor \tag{1}$$

This will then give the maximum number of feature vectors that can fit within a singular S_{io}. Then, to find the necessary amount of cluster leaders l, which would fit all n vectors, one must determine this by utilizing the S_c value. This gives the equation:

$$l = \left\lceil \frac{n}{S_c} \right\rceil \tag{2}$$

[1] E.g., see: https://git.savannah.gnu.org/cgit/coreutils.git/tree/src/ioblksize.h

The third method to ensure efficiency and accuracy is the creation of an index tree of cluster leaders in the eCP. When the l cluster leaders have been determined, the actual structure of the index tree can be chosen and created. The eCP has a height parameter L, which is set at indexing time, that determines how the index should be structured internally.

Each level in the tree is based on a similar clustering approach as with the feature vectors. Each leader in the index structure represents on average

$$S_n = \sqrt[L]{l} \tag{3}$$

leaders below itself, which are either actual cluster leaders or leaders of internal nodes. The process works by first selecting l/S_n group leaders from the collection of cluster leaders, and assigning each cluster leader to the nearest group leader, thus grouping the l cluster leaders into groups of (on average) S_n cluster leaders. These internal nodes are then also grouped into groups of S_n leaders; this process continues recursively until the top internal group has fewer than S_n leaders and becomes the root of the index tree.

The L parameter indirectly controls how large the internal nodes should be in the structure. A too-large L value would lead to many small internal nodes in the structure, which would incur a large overhead cost and a loss of result quality [2], while a too-small L would create large internal nodes leading to worse performance, as this would cause more comparisons performed within each internal node during query processing.

3 Index Maintenance Strategy

We now consider the scenario where additional vectors are inserted into the eCP index after the initial index construction has completed. The goal of any index maintenance strategy should be that query processing is affected as little as possible both in terms of efficiency and accuracy. Consider a strategy where these vectors are simply inserted into the appropriate clusters without updating the local structure in the index. The clusters would eventually become overly large, leading to an increase in query processing time and loss of accuracy. Therefore it is paramount that the clustering is dynamically maintained along with the index of cluster leaders in order to keep the cluster sizes and the distribution of data as stable as possible.

A second goal of an index maintenance strategy is that the index maintenance itself should be efficient. This goal in turn implies two things: a) that the insertion operations should be aggregated in order to avoid the costs associated with repeatedly updating the same clusters and b) that the structure of the index itself should be maintained, rather than rebuilt. Therefore it is necessary to perform local maintenance of the index in order to avoid having to make global changes to the clustering and the index structure.

The main topic of this section is a proposal for such a maintenance strategy. In Section 3.1 the procedure of aggregating insertions is detailed and in Section 3.2 the maintenance strategy for the cluster index is described. A cost model for this maintenance strategy is then developed in Section 3.3 and then the potential impacts on the effectiveness of the eCP is briefly outlined in Section 3.4.

3.1 Insertions

When a cluster is chosen for insertions in eCP, there are two different scenarios that can happen in terms of IO operations. The first scenario is that the insertion is performed directly to disk, which means that the insertion operation will require a disk read and a disk write operation in order to update the chosen cluster with the inserted data. Since this cluster may be anywhere on the disk, this will incur a random read operation and a random write operation.

Alternatively, insertions could be buffered, where a given number of inserted feature vectors is buffered in RAM. When the buffer is then filled, some (or all) of the buffered data must be written to disk. In this case, multiple insertions to the same cluster would only require one disk read and one disk write, thus saving some IOs compared to direct insertions.

3.2 Index Maintenance

In order to maintain the index, the index structure must grow. This implies firstly that L cannot be a static parameter. Instead, the average size of the internal nodes S_n is given as a parameter and used to determine a suitable L, by reversing Equation 3:

$$L = \lceil \log_{S_n}(l) \rceil \tag{4}$$

As the size n of the data collection grows, then so does l and therefore L will also eventually grow, resulting in a deeper index. Based on previous results with eCP, a suitable value for S_n is about 100 [4].

A dynamically growing index implies secondly that we should over-allocate the number of cluster leaders, to leave free space in the clusters for insertions. We propose using a new percentage parametrer lo, such that clusters (and internal nodes) are only filled to lo capacity; based on industry experience with index maintenance, a decent lo value is about 70% of full capacity. Likewise, to avoid aggressively reorganizing clusters, we propose using a corresponding hi parameter, which is used to determine when to re-cluster a part of the index tree. Together, the two parameters help to avoid frequent local re-clustering.

The index maintenance strategy we propose is indeed based on local reorganization of a sub-tree in the index. Consider a set of cluster leaders, which are grouped together under the same internal node. If an insertion into any of these clusters causes the average cluster size in the group to grow beyond $hi \times S_c$, then all of the clusters in the internal node are re-clustered together.

This re-clustering process works by determining the number of feature vectors represented in that internal node, by summing the number of feature vectors in each cluster in the group; this sum is called n^*. Then the number of clusters l^*, which should be represented in the internal node, is determined using a variant of Equation 2, modified to use n^*:

$$l^* = \left\lceil \frac{n^*}{\lfloor S_c \rfloor} \right\rceil \tag{5}$$

The l^* cluster leaders are selected randomly and the feature vectors are assigned to them, as in the original clustering process. These new l^* cluster leader vectors now form the updated internal node.

It is possible that the internal node grows so much that the parent node now has more than $hi \times S_n$ nodes on average. Then the parent node can also be reorganized in the exact same manner, by determining how many internal nodes should be used and grouping the representatives into these new internal nodes. This process can propagate all the way to the top of the index structure; if the root node grows to have more than $hi \times S_n$ children, then it must be split using the same reorganization process with a new root node created at the top of the tree; when this happens, the height of the tree L grows.

There are some important properties of this process worth noting. First, every re-clustering is local, as only one internal node and its children are reorganized at a time. The index maintenance therefore only flows upwards in the index tree and never downwards. As a result, the only clusters of feature vectors, that are affected, are the ones that originally caused the re-clustering that then propagated up the tree. Second, the reorganization of internal nodes can cause clusters to move within the tree to a closer parent node, which then will make the cluster more likely to be correctly found during the search process. This strategy should therefore maintain overall result quality over time.

3.3 Cost Model

We now describe a simple cost model for this strategy, that we will use to evaluate the efficiency of the strategy. As previously discussed, we assume each disk operation reads or writes $S_{io} = 128$ KB. The IO operations may be either sequential or random in nature, and we denote the costs of the operations C_{SR} and C_{SW} for sequential reads and writes, respectively, and C_{RR} and C_{RW} for random reads and writes.

Direct insertions of a feature vector into a cluster requires reading of that cluster from disk, adding the new feature vector, and then writing the cluster back to disk. Assuming that the cluster contains n feature vectors, the cost of an insertion is:

$$C_I = \lceil n/S_c \rceil * C_{RR} + \lceil (n+1)/S_c \rceil * C_{RW} \tag{6}$$

When a feature vector is inserted into the aggregation buffer, no cost is assigned. Once the buffer is full, in our cost model a full flush of the insertion buffer is forced. We model two different types of flushes, where (a) individual clusters are read as needed and updated, with the same cost as above, or (b) where all the clusters are read and written sequentially, resulting in a cost of $l \times (C_{SR} + C_{SW})$.

Table 1. IO latencies for the two devices modelled in our experiment.

Device	C_{RR}	C_{RW}	C_{SR}	C_{SW}
Toshiba 7200RPM HDD	4.930 ms	2.110 ms	0.642 ms	0.645 ms
Intel P3700 PCI-E SSD	0.055 ms	0.122 ms	0.056 ms	0.121 ms

If a cluster forces a re-clustering, the leader will take all of its l_n clusters and their n^* descriptors and re-cluster these into l_n^* new clusters. The cost associated with such a re-clustering is modelled as:

$$C_R = l_n \times C_{RR} + l_n^* \times C_{RW} \tag{7}$$

In the case where a internal node forces a re-clustering of its parent, the parent node will then take its l_n representatives and reorganize them into l_n^* new leader groups. Assuming that each node fits within one disk IO, the cost of such reorganization is also given by Equation 7. The cost of reorganizing the root is similar.

3.4 Discussion

We have proposed a new insertion strategy for the eCP high-dimensional index. As the analysis of the next section shows, the strategy achieves our two efficiency goals: the strategy itself can be implemented efficiently and it leads to a balanced cluster size distribution, which is key to the retrieval efficiency. We have also argued that because both feature vectors and clusters can dynamically move within the index tree, the result quality is likely to be maintained. Testing this latter hypothesis is part of our future work.

4 Experiments

In order to explore the performance of the index maintenance strategy, we have implemented a simulation model, which is a modified edition of the one used in previous experiments with the NV-tree [6,8].

The simulator starts by instantiating an index in an initial state, with a given number of feature vectors. Then the simulator inserts vectors into the index, using the index maintenance strategy, as long as desired. During this process the simulator uses the cost model above to keep track of the total IO cost.

For the experiment reported below, an IO size of S_{io} =128 KB is used. The initial index contains 50 million feature vectors, with 1.5 billion subsequent insertions performed to measure efficiency. We consider disk IO latencies for the two devices presented in Table 1, which represent a competitive HDD and SSD, respectively [6]. We model SIFT feature vectors (s_v = 132 bytes) and set the internal node parameter $S_n = 100$, which has been shown to give good results [4]. The simulator can model both types of insertion buffer flushes, but for eCP a full scan of the index was more efficient.

Table 2. Simulation results: Total time to insert 1.5 billion feature vectors.

Index	**HDD**	**SSD**
NV-tree	895.3 hours	2.8 hours
eCP	396.4 hours	20.6 hours

4.1 NV-tree

As a baseline, we compare to the NV-tree, which was already implemented in the simulator [6]. The NV-tree is a tree-based high-dimensional index, which utilizes a combination of projection of data points along random lines, and partitioning of the projected space, to separate the dataset into partitions which are designed to fit within a single IO. In order to maintain the index while insertions are performed, the NV-tree must select new random lines and re-project the data contained within its partitions, in order to maintain the small partitions.

The NV-tree only stores the feature vector identifier in the leaves of the tree. To re-project feature vectors, they must therefore be retrieved from disk separately. For an SSD, the most efficient way to do this is simply to issue many small reads for the feature vectors, as small random reads are efficient with SSDs. For an HDD, however, the NV-tree must maintain an auxiliary data-structure, called partition files, which contain the feature vectors for each partition. This leads to significant additional cost of maintaining this auxiliary structure.

4.2 Results

Table 2 shows the estimated time for inserting 1.5 billion feature vectors into both the eCP index and the NV-tree index. Overall, the results show that the newly proposed strategy for eCP index maintenance is competitive; with a HDD, eCP outperforms the NV-tree, while the NV-tree performs better on an SSD.

The reason why eCP performs better for the HDD is due to the overhead for the NV-tree, which has to maintain both the index and the partition files separately during re-projection maintenance, which is very costly in terms of IOs. Meanwhile, the eCP is a simpler index which stores the features within the index, which is acceptable on an HDD. With eCP, nearly 1 billion disk operations are issued (20% are random reads, 40% are sequential reads, and 40% are sequential writes) while for the NV-tree, about 5 billion operations are issued (evenly split between sequential reads and writes). The reason why the NV-tree is more efficient than eCP on an SSD, is more complex. Even though the NV-tree still requires more disk operations (about 3.2 billion), almost all these operations are small reads, which are very efficient on an SSD.

We also considered the cluster size distribution for eCP. After inserting 1.5 billion vectors, the smallest cluster contained 688 vectors and the largest 1083 vectors, with an average of 894 vectors. As the intended average cluster size is $S_c = 992$ vectors, these results indicate that the strategy will maintain a good cluster size distribution.

5 Conclusion

In this paper we argued for the importance of dynamic maintenance of high-dimensional index structures. We presented a novel index maintenance strategy for the scalable eCP index structure. This strategy aims at maintaining the balanced cluster sizes, which are crucial for the eCP to maintain its disk performance, while also implementing local reorganizations of clusters to reduce maintenance cost and preserve the accuracy of the index. We have implemented this strategy in a simulation model and compared it to the very efficient NV-tree structure, showing that while index maintenance of the NV-tree is more efficient with SSDs, the new index maintenance strategy for eCP is nevertheless quite competitive, with only ≈ 21 hours required to insert 1.5 billion feature vectors.

References

1. Babenko, A., Lempitsky, V.: The inverted multi-index. In: Proc. Conference on Computer Vision and Pattern Recognition (CVPR). pp. 3069–3076. Providence, RI, USA (2012)
2. Chierichetti, F., Panconesi, A., Raghavan, P., Sozio, M., Tiberi, A., Upfal, E.: Finding near neighbors through cluster pruning. In: Proc. Symposium on Principles of Database Systems (PODS). pp. 103–112. Beijing, China (2007)
3. Fagin, R., Kumar, R., Sivakumar, D.: Efficient similarity search and classification via rank aggregation. In: Proc. ACM SIGMOD International Conference on Management of Data. pp. 301–312. San Diego, CA, USA (2003)
4. Guðmundsson, G.Þ., Jónsson, B.Þ., Amsaleg, L.: A large-scale performance study of cluster-based high-dimensional indexing. In: Proc. Workshop on Very-Large-Scale Multimedia Corpus, Mining and Retrieval (co-located with ACM Multimedia). Firenze, Italy (2010)
5. Jegou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor search. IEEE Transactions on Pattern Analysis and Machine Intelligence **33**(1), 117–128 (2011)
6. Jónsson, B.Þ., Amsaleg, L., Lejsek, H.: SSD technology enables dynamic maintenance of persistent high-dimensional indexes. In: Proc. ACM International Conference on Multimedia Retrieval (ICMR). pp. 347–350. New York, NY, USA (2016)
7. Lejsek, H., Ásmundsson, F.H., Jónsson, B.Þ., Amsaleg, L.: Transactional support for visual instance search. In: Proc. Similarity Search and Applications (SISAP). pp. 73–86. Lima, Peru (2018)
8. Ólafsson, A., Jónsson, B.Þ., Amsaleg, L., Lejsek, H.: Dynamic behavior of balanced NV-trees. Multimedia Systems **17**, 83–100 (2011)
9. Sigurðardóttir, R., Hauksson, H., Jónsson, B.Þ., Amsaleg, L.: Quality vs. time tradeoff for approximate image descriptor search. In: Proc. IEEE EMMA Workshop (co-located with ICDE). Tokyo, Japan (2005)
10. Sivic, J., Zisserman, A.: Video google: A text retrieval approach to object matching in videos. In: Proc. IEEE International Conference on Computer Vision (ICCV). pp. 1470–1477. Nice, France (2003)
11. Tao, Y., Yi, K., Sheng, C., Kalnis, P.: Quality and efficiency in high dimensional nearest neighbor search. In: Proc. ACM SIGMOD International Conference on Management of Data. pp. 563–576. Boston, MA, USA (2009)