RCached-tree: An Index Structure for Efficiently Answering Popular Queries

Manash Pal
Dept. of Computer Science and Engineering,
Indian Institute of Technology, Kanpur, India.
manashp@cse.iitk.ac.in

Arnab Bhattacharya
Dept. of Computer Science and Engineering,
Indian Institute of Technology, Kanpur, India.
arnabb@iitk.ac.in

Debjyoti Paul
Dept. of Computer Science and Engineering,
Indian Institute of Technology, Kanpur, India.
debpaul@cse.iitk.ac.in

ABSTRACT

In many applications of similarity searching in databases, a set of similar queries appear more frequently. Since it is rare that a query point with its associated parameters (range or number of nearest neighbors) will repeat exactly, intelligent caching mechanisms are required to efficiently answer such queries. In addition, the performance of non-repeating and non-cached queries should not suffer too much either. In this paper, we propose *RCached-tree*, belonging to the family of R-trees, that aims to solve this problem. In every internal node of the tree up to a certain level, a portion of the space is reserved for storing popular queries and their solutions. For a new query that is encompassed by a cached query, this enables bypassing the traversal of lower levels of the subtree corresponding to the node as the answers can be obtained directly from the result set of the cached query. The structure adapts itself to varying query patterns; new popular queries replace the old cached ones that are not popular any more. Queries that are not popular as well as insertions, deletions and updates are handled in the same manner as in a general R-tree. Experiments show that the RCached-tree can outperform R-tree and other such structures by a significant margin when the proportion of popular queries is 20% or more by reserving 30-40% of the internal nodes as cache.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query Processing; H.2.2 [**Physical Design**]: Access Methods

Keywords

RCached-tree, Caching, Similarity Search, Popular Queries

1. INTRODUCTION

The success of R-trees [5] as an index structure in the realm of database similarity searching has been phenomenal. Since it is a height-balanced structure with the objects stored only at the leaf levels, when the result sizes are comparable, the performance of different such queries tend to be similar. Each query is agnostic of the general trend of querying, and is solved afresh irrespective of how "popular" the query is, without resorting to any memory or "caching" of previous queries and their results.

In many cases, however, the situation is different. Consider a city with designated public parking spots stored as objects. In general, queries for parking spots are distributed along the entire city. However, consider the situation just before the start of an important sports event or a cultural event. Most queries on finding suitable parking spots will be near the venue, i.e., the query points are no more distributed along the entire city, but are concentrated around a small area in it.

An R-tree will not treat these queries as any special, and will process them in exactly the same manner as any other query. Intuitively, it makes sense to "remember" the location and answer of previous queries in such situations where similar queries re-appear. The above specification immediately points to a caching solution.

However, it is unlikely that *exactly* the same query will be repeated. Either the query location will change, or its attribute (such as the range or the number of nearest neighbors) will change, or, as is most likely, both the parameters will slightly differ. Hence, a simple hashing of query parameters (and results) is unlikely to help much. Thus, there needs to be a mechanism to identify "similar" queries, and then process them using an index structure. Moreover, the performance of non-repeating and non-cached queries should not suffer too much either.

In this paper, we propose the RCACHED-TREE for this purpose. It is a member of the R-tree family of height-balanced hierarchical disk-aware structures. It works similar to an R-tree for "non-popular" queries, i.e., it traverses the relevant paths of the tree and accesses the corresponding nodes to retrieve the answer. However, when a query is deemed "popular" in a subtree, its results are stored in the node so that accessing its children is bypassed, and the answer set for this subtree is retrieved directly.

For a node, some particular proportion of the storage for index entries is used as the "cache". Although this reduces the amount of indexing that can be achieved as the number of children nodes are reduced, it provides a short-cut to the answer for popular queries, thereby essentially bypassing all disk accesses for levels below the current one.

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.

ACM 978-1-4503-2263-8/13/10 ...$15.00.
http://dx.doi.org/10.1145/2505515.2507817.

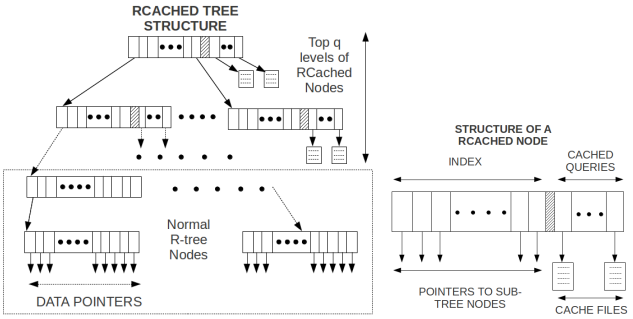

Figure 1: Structure of a RCached-tree and its node.

Our main contribution in this paper, therefore, is to introduce the concept of *result caching* in an R-tree.

Note that we also process the non-popular queries efficiently as in that case, searching through the normal R-tree structure takes over. This is an important point as most of the hashing-based solutions will either suffer for such queries or will need to maintain a separate indexing structure to which the non-popular queries will be delegated to.

2. RELATED WORK

Traditional methods of utilizing result sets from previous similar queries are that of hashing. While static hashing does not scale well, dynamic hashing techniques such as [4, 7] work only for point queries and cannot answer similarity search queries. The disk-based hashing methods do not explore the spatial locality well, except LSH [6], which, however, is not efficient for arbitrary range and kNN queries.

For disk-based databases, R-tree [5] remains the quintessential data structure for spatial indexing. It uses the concept of minimum bounding rectangles (MBRs) to cluster data points into hierarchy of regions. R*-tree [2] and X-tree [3] are some important variants of the R-tree.

Most nodes in disk-aware index structures are rarely full; while the space utilization in B+-tree is only 67% [8], that in R-trees is in the range of only 50% to 70% [2].

CSB+-trees made B+-trees cache aware by creating more space for index entries in a node after replacing pointers by a single entry to an array of pointers [9]. Extra memory buffer was added to each node in [1] to solve queries in a lazy manner. Lower levels of the tree are not accessed unless the query buffer is full at a given level. Hence, query processing occurs in batches thereby saving random disk I/O. In [11], pointer to frequently accessed leaf nodes are stored in the unused part of a B+-tree internal node to reduce the overall node access time.

3. THE RCACHED-TREE

In this section, we describe the RCached-tree in detail.

3.1 Structure of a Node

Each node of the R-tree is divided into two parts: (i) the *index* part which contains pointers to children nodes, and (ii) the *cache* part which stores (pointers to) results of popular queries. The index part of the node contains index keys (MBRs) as in an R-tree [5]. The percentage of space in a node earmarked for the cache part, denoted by p (at most 50%), is a design parameter which remains fixed for a tree.

Since part of the node is utilized for caching, the number of keys for indexing decreases which may result in an overall increase in the height of the tree. Analysis (omitted due to space restrictions) shows that in most real datasets, the height differs by at most 1. If the fanout of a normal R-tree is β, then that of an RCached-tree is at least $\beta/2$.[1]

Each cache entry in a node consists of the following four fields: (i) Query point Q: The d-dimensional point in the data space. (ii) Parameter s: Range value for range query or number of nearest neighbors for KNN query. (iii) Pointer ptr: Disk pointer where the result set (called "cache file") for the query is stored. (iv) Popularity pop: The popularity measure of the query at this node.

3.2 Structure of RCached-tree

Intuitively, it makes little sense to introduce caching beyond a point. The reason is that since the lower level nodes are quite close to the leaf, accessing the cache via random disk accesses may not save any more than directly accessing the leaves. Moreover, the extra overhead of writing cache files at this level may actually degrade the performance. Hence, we introduce an engineering optimization and retain caching only up to some top q levels from the root. The rest of the levels have fanout equal to that of the corresponding R-tree. For experiments, we fixed q at 3 since the trees were of height either 4 or 5.

The RCached-tree, therefore, is a hybrid structure with the top q levels having a fanout of at most α while the lower levels can have a fanout up to β. Figure 1 shows the general structure of a RCached-tree. We next describe how such a tree with a non-uniform branching factor is constructed.

3.3 Construction

First, a normal R-tree with a fanout of β is built. The MBRs within the nodes at the q^{th} level of this R-tree then act as objects for the re-construction of the above part of the tree with a fanout of α. Essentially, the top q levels of the R-tree are discarded and are replaced by $q + \delta$ levels of RCached-tree nodes. The increase in height of the tree is, therefore, δ, which, as discussed earlier, is at most 1.

3.4 Insertion and Deletion

Insertions and deletions are handled in the same way as in an R-tree [5]. When the MBR of a node and/or its children are changed, the result sets of the cached queries become corrupt. We then simply mark those contents as "dirty" and do not use them any further. Essentially, they are treated as purged and, thus, the consistency of the operations as long as they are sequential, are maintained.

As in most caching solutions, the structure works the best when there are less updates. A dataset having a large amount of insertions and deletions after the queries start arriving is unlikely to benefit much from caching. Therefore, we assume that updates are relatively infrequent.

4. SIMILARITY SEARCH QUERIES

We describe the search for a query Q (which includes the query location and the associated parameters such as range or number of nearest neighbors) over a node t. Query processing begins at the root.

If the node t is beyond the level where caching is used, the normal R-tree search is performed. Otherwise, first the

[1]Assuming a total of 2^{21} entries and $\beta = 128$, the heights are respectively 3 and 4. For 2^{22} entries, they are both 4.

cache entries of t are searched. If there exists a cache entry C that completely encompasses the bounding box of the query Q, then the result set is collected as a subset of objects from the cache file of C by actually computing the distances of those objects from the query.

Otherwise, if no such cache entry is present, then all the subtrees that satisfy the query predicate are examined in a depth-first-search (DFS) order. A track (and copy) of all nodes up to the caching level touched by this query are kept in the memory. This is done so that when the query finishes, the query can be injected as a cache entry in these nodes.

If the cache slots are full, the new query has to compete with the existing cache entries for a place using the popularity measure as described later in Section 4.3. When a cache slot is free, then of course, the query wins the free slot. When a query wins a slot, a cache entry is made corresponding to the query, and its results pertaining to the subtree is written back to disk as the cache file.

4.1 Random Queries

For random queries, the above cache file creation on disk creates unnecessary random I/O operations as it is unlikely that the cache file will be ever used again. Thus, to guard against this problem, when a query arrives for the first time, even if it wins a cache slot (or if there is a free cache slot), the corresponding result set is *not* written as a cache file. However, when the query arrives the next time (i.e., when there is a hit for the cached query), the results are written to the disk as a cache file.

4.2 kNN Queries

For a kNN query, the results are written only for those nodes at which the search has fetched k new entries within that sub-tree (effectively computing the kNN radius for the query within the sub-tree). At each node, a new query Q' updates its current kNN radius $Q'.r$ to $\min\{d(Q,Q') + Q.r, Q'.r\}$ for all valid cache entries Q since this ensures that it will find k new values within that range to compare with. The query execution for Q' within this subtree thus achieves superior pruning by reducing the radius further.

The root node is treated differently since it is unlikely that a new query finds a useful cache entry as it always differs slightly. To solve this, we use Theorem 4 of [10]. When applied in this context, it implies that the k-NN of a new query Q' is contained within the m-NN of a cached query Q if and only if $2d(Q,Q') \leq d(Q, mNN(Q)) - d(Q, kNN(Q))$, where $k < m$ (the distance function used must be a metric).

Thus, instead of searching for k nearest neighbors, the RCached-tree searches for m nearest neighbors at the root where $m > k$. For a new query Q', it checks whether the above inequality holds for a cached query Q. If yes, the result set of mNN of Q is directly used to answer the kNN of Q'. Otherwise, the normal search procedure is continued.

When m is large, the query time increases as more nearest neighbors are searched. When m is small and close to k, then the tolerance for the distance of the new query to the cached query decreases. We choose $m = 1.2 \times k$.

4.3 Cache Popularity Measure

The simple model of assigning the number of times a query has arrived as its popularity does not work as then it is harder for a new query that is becoming popular to replace an old query that had been popular earlier. Thus, the popularity function should be such that newer queries gain popularity quickly while old cached queries lose popularity if it has not been used in a long time.

Without loss of generality, we assume that a query arrives at every time tick. Thus, only the frequency of queries, and not the actual wall clock time, is important. To measure the popularity of a query, we consider a time-interval of M past queries, i.e., M denotes the time history beyond which the appearance of a query does not affect its popularity. We use $M = 10000$ for our experiments.

Suppose, a query occurs at the current time t and uses a particular cache slot C. Also suppose, the last time the same cache slot C was used was at time t'. We denote the time difference by $\Delta t = t - t' > 0$. We use the concept of *expected cache usage difference*, η, that signifies the amount of time within which a cache slot should be re-used. We fix η at $0.1M = 1000$. The parameter η is the expected time within which a popular query should re-appear. Lower values indicate that popular queries are more frequent.

The function $\Psi(\Delta t) = \Delta t - \eta$ measures the difference of the usage of this cache slot with the expected value. If $\Psi < 0$, the cache slot is used more frequently and should gain popularity. On the other hand, a positive value of Ψ implies that the cache slot is getting unpopular and, hence, its popularity should decrease.

If the original popularity value of the cache slot C at time t' was Pop', then the new popularity value of C at time t is

$$Pop(C,t) = \rho\left(\rho^{-1}(Pop') + \alpha\Psi(\Delta t)\right) \quad (1)$$

The function $\rho(x)$ measures the popularity value of x:

$$\rho(x) = (-x/c)/(1 + |x/c|) \quad (2)$$

where c is a normalization constant whose value is $M/10$. The function $\rho(x)$ has the range $(-1, +1)$ with $\rho(0) = 0$. It is also invertible. The parameter α controls the rate at which a query changes its popularity. We fix it to 0.5.

5. EXPERIMENTS

5.1 Experimental Setup

The experiments were performed on a Linux machine running Ubuntu 11.04 with 2.6.38-16-generic kernel. The system has 7.8 GB of main memory and 8 Intel(R) Core(TM) i7-2600 CPU processors of frequency 3.40 GHz.

We compare the performance of RCached-tree against R-tree (with quadratic split) and X-tree based on the following parameters (the default values are in parentheses): dimensionality (3), dataset cardinality (10^6), popularity density (30%), data type (uniform), page size (8 KB), range value (0.01% of the largest possible distance), number of nearest neighbors (20), and percentage of node used for caching purposes (30%). The real data is that of San Francisco's road network (1,74,955 nodes), available from `http://www.cs.fsu.edu/~lifeifei/SpatialDataset.htm`.

For each set of parameters we executed 50,000 queries divided into 5 slots of 10,000 queries each. For each such set, we perturb both the query centers and the range radii by a small amount while we keep the k fixed. We considered that at a given time, there are 10 regions of popularity. Each slot of 10,000 queries has a different set of 10 popular regions. Thus, our results indicate the ability of our proposed structure to adapt to changes in popular regions over time. To measure the performance variation for each parameter, we

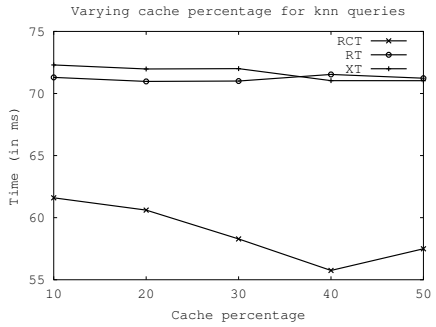

Figure 2: KNN queries: Cache percentage.

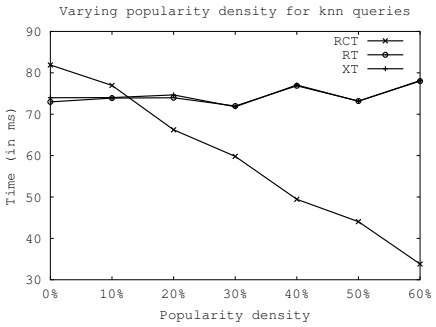

Figure 3: KNN queries: Popularity density.

vary only that and keep all the others at their default values for both range and kNN queries. (Due to lack of space, we only show representative results.)

5.2 Results

When the cache percentage is low, there is more competition for a cache slot, and hence, the query performance degrades due to repeated purging of the cache. On the other hand, when it is too much, the space for indexing goes down, and consequently, the queries that are not popular suffer as then more nodes may need to be searched. Figure 2 reveals that the running time is minimum around 40%.

When popularity density of queries increases, the running time decreases steadily for both kNN (Figure 3) and range queries. Even with only 10% popular queries, the performance is comparable with that of R-tree and X-tree.

The performance of all the structures improves with increasing dimensionality for low to medium values (this and all subsequent graphs are omitted). When dimensionality is low, the branching factor is too high, and the amount of CPU time spent in searching through the contents of a node dominates the total running time. At higher values, the curse of dimensionality eventually kicks in. With increasing data cardinality, the comparative advantage over R-tree and X-tree increases, thereby exhibiting the better scalability of RCached-tree. Due to increase in answer set size, the time increases with the number of nearest neighbors and range for kNN and range queries respectively. When page size increases, the CPU time required to process a page as well as the time for sequential I/Os increase. Consequently, the running time suffers. A uniform dataset performs worse than a more concentrated dataset (e.g., Gaussian) due to the high degree of overlap between the internal nodes. The average for the real dataset is the best due to its smaller size and is within 20 ms for the RCached-tree.

5.3 Analysis of the Results

Analyzing the different experimental results, we can draw the following conclusions: (1) The height of the RCached-tree is never more than 1 more than the original R-tree. Thus, even if there are no popular queries, the performance is comparable to that of an R-tree. (2) The performance of a RCached-tree steadily improves with increase in number of popular queries. When the proportion of popular queries is more than 20%, the RCached-tree clearly outperforms R-tree and X-tree. (3) The RCached-tree adapts itself nicely and quickly to the changing query distribution. This is an important feature in real applications where such shifts in queries are common. (4) The cache index of a node should be around 30-40%. Otherwise, there is not enough cache space, and repeated purging of popular queries degrades the performance. (5) The RCached-tree performance is robust across different values of number of nearest neighbor queries, the range of the query, the page size and the dimensionality. This makes the structure well applicable to different situations without the need for extensive parameter tuning.

6. CONCLUSION

In this paper, we have proposed a new disk-based hierarchical indexing structure, the RCached-tree, which efficiently solves popular similarity search queries by caching the results of similar queries. For non-popular queries, the standard R-tree indexing mechanism is used to solve them efficiently. Experiments demonstrated the scalability and practicality of our structure. In future, we would like to improve the update handling mechanism. Further, adopting optimizations from other methods such as index compression, lazy querying, etc. remain interesting avenues of work.

7. REFERENCES

[1] L. Arge, K. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. In *ALENEX*, pages 328–348, 1999.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, 1990.

[3] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *VLDB*, pages 28–39, 1996.

[4] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing: A fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979.

[5] A. Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, 1984.

[6] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

[7] W. Litwin. Linear hashing: A new tool for file and table addressing. In *VLDB*, pages 212–223, 1980.

[8] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2007.

[9] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. *SIGMOD Rec.*, 29(2):475–486, 2000.

[10] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD*, pages 79–96, 2001.

[11] C. Yu, J. Bailey, J. Montefusco, R. Zhang, and J. Zhong. Enhancing the B+-tree by dynamic node popularity caching. *Inf. Process. Lett.*, 110(7):268–273, 2010.