Similarity Searching: Towards Bulk-loading Peer-to-Peer Networks

Vlastislav Dohnal  Jan Sedmidubský  Pavel Zezula  David Novák
Faculty of Informatics
Masaryk University
Botanická 68a, Brno, Czech Republic
{dohnal, xsedmid, zezula, david.novak}@fi.muni.cz

Abstract

Due to the exponential growth of digital data and its complexity, we need a technique which allows us to search such collections efficiently. A suitable solution seems to be based on the peer-to-peer (P2P) network paradigm and the metric-space model of similarity. During the building phase of the distributed structure, the peers often split as new peers join the network. During a peer split, the local data is halved and one half is migrated to the new peer. In this paper, we study the problem of efficient splits of metric data locally organized by an M-tree and we propose a novel algorithm for speeding the splits up. In particular, we focus on the metric-based structured P2P network called the M-Chord. In experimental evaluation, we compare the proposed algorithm with several straightforward solutions on a real network organizing 10 million images. Our algorithm provides a significant performance boost.

1. Introduction

Current data processing applications often use data with considerably less structure and search the data with much less precise queries than traditional database systems. An example is multimedia data like images or video clips that offer query-by-example search. This situation is what has given rise to similarity searching. The most general approach to similarity search, still allowing construction of index structures, is modeled by a metric space. Many metric-based index structures were developed and surveyed recently [25, 30]. The latest efforts in this area focus on the design of distributed access structures which exploit more computational and storage resources [2, 13, 4, 3] in order to cope with the problem of exponential growth of the data to be processed. A suitable solution arises from peer-to-peer (P2P) networks which are scalable.

In this digital-explosion age, a real P2P-based data structure organizes non-trivial amounts of data, e.g., tens of millions of data objects (records) or even more. In this respect, each peer of this network has to also maintain a large number of data objects, so the search within the peer is usually sped up by a local (centralized) index structure. During the life cycle of the P2P structure, peers are split as the data collection grows and new peers join the network. Consequently, the data content of the overloaded peer must be halved, that is the local index structure must be split.

In this paper, we focus on effective splits of local metric-based indices. Namely, we study this problem on the P2P index structure called the M-Chord [22] which uses the M-tree [8] to index peer’s local data. This M-tree is enriched with extensions of the Slim-tree [28] and the PM-tree [27]. The paper is structured as follows. After related work, Section 2 contains description of the architecture of our image retrieval system. In Section 3, we present four algorithms to split peers. The paper concludes with a performance comparison.

1.1. Related work

Multimedia data is often modeled by high-dimensional spaces, so a variety of high-dimensional index structures were proposed, e.g., the R-tree, the TV-tree, the SS-tree or the X-tree. A more generic approach to index such data is to exploit the metric space model, e.g., the M-tree [8], the Slim-tree [28], the GNAT [6], the SAT [21], or the D-index [11].

Due to the need to organize large databases, many techniques to bulk-load index structures were proposed. The Hilbert R-tree [17] is probably the first modification of R-tree aiming at optimized insertion of a large number of data records. It is based on sorting the elements first and then building the tree bottom-up. Earlier, similar techniques were proposed for the B-tree [18] and the quadtree [23, 15], for illustration. A bulk-loading algorithm which also optimizes the tree in order to improve search efficiency is proposed in [5]. It is demonstrated on the X-tree but it can be applied to any R-tree-like structure. Another variant of R-
tree called the Priority R-tree [1] is slightly less efficient in bulk-loading than the Hilbert R-tree, however, the resulting tree has better query performance. A bulk-loading strategy which sorts the data items first cannot be applied to structures based purely on the metric space model because there is no linear ordering defined in the general metric space. A generic bulk-loading algorithm that does not sort new items beforehand, is presented in [10]. This proposal was further improved in [9] and applied to the Slim-tree. For the original M-tree, two bulk-loading algorithms were proposed. The first one [7] decreases the construction costs, while retaining the same query performance. The second one [26] focuses on improving query performance without considering building costs. In [14], a general algorithm for bulk-loading applicable to height-unbalanced structures is proposed.

In parallel environments, the problem of bulk-loading was studied for GridFiles [20, 19]. Papadopoulos and Manolopoulos investigated this problem in parallel database systems [24]. To the best of our knowledge, the bulkloading operations have not been explored in the area of structured peer-to-peer systems yet mainly from the splits of peers point of view.

1.2. Contributions

In this paper, we propose a novel strategy suitable for any peer-to-peer network which models data as a metric space and uses the M-tree to index the peer’s content. This strategy is utilized when a peer is split as the dataset grows and new peers join the network. We propose a sophisticated algorithm which optimizes the process of splitting. This algorithm is compared with other algorithms detailed in the paper. In addition, we enhance the M-Chord with this sophisticated algorithm and compare it with the original version.

2. The similarity search system

The retrieval system studied in this work is a peer-to-peer index structure called the M-Chord [22] which organizes data modeled as a metric space. The core idea of the M-Chord is to use a modification of the method called the iDistance [29] to map the metric space into a one-dimensional domain. The standard peer-to-peer protocols like Chord or Skip Graphs are then used to divide the transformed data domain among the peers and to provide navigation within the system. Each peer organizes its assigned data in the M-tree. We use an extended version of M-tree which encapsulates (1) the node-split algorithm based on building a spanning tree taken from the Slim-tree and (2) filtering using a set of independent pivots taken from the PM-tree.

Figure 2. Examples of partitioning: (a) the ball partitioning and (b) the generalized hyperplane partitioning.

2.1. Metric space fundamentals

Formally, the metric space is a pair $M = (\mathcal{D}, d)$, where $\mathcal{D}$ is the domain of objects and $d$ is the total distance function satisfying the metric postulates: $d(x, y) \geq 0$ (non-negativity), $d(x, y) = d(y, x)$ (symmetry) and $d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality). The distance function expresses the dissimilarity of two objects. Following the query-by-example paradigm, the range query $R(q, r)$ retrieves all objects $o$ within the range $r$ from $q$ ($d(q, o) \leq r$), and the nearest neighbors query $k\text{NN}(q, k)$ returns the $k$ objects with the smallest distances to $q$.

2.2. M-tree

The M-tree [8] is a dynamic balanced tree-like structure. Similarly to B-trees, it is built in a bottom-up fashion by splitting over-filled nodes. The M-tree is a $k$-ary tree, so every internal node consists of up to $k$ entries. Besides the reference to a subtree, the entry contains a pivot $p$ (selected object) and a covering radius $r^c$ which together define a ball-like region $R(p, r^c)$. This region covers all objects stored within the subtree. The leaf nodes store data objects accompanied with their distances to the pivot in the parent node. The internal nodes keep distances to the parent node’s pivot as well. All these distances are utilized in order to prune some tree branches by search/insert/delete algorithms. An illustrative example of M-tree is shown in Figure 1. The ball regions of the leaves and the internal nodes are identified by dashed circles and solid circles, respectively.

3. Splitting the M-tree

In this section, we analyze the problem of splitting a peer of P2P network, where the peer organizes its data in the M-tree. In general, partitioning principles are based either on ball partitioning or generalized-hyperplane partitioning (see...
Figure 1. Example of an M-tree consisting of three levels. Above, a 2-D representation of clustering. Pivots are denoted by crosses and the circles around pivots correspond to the values of covering radii.

Figure 2). Though there are other partitioning policies, they build on these two basic variants in principle. The algorithms presented below can be applied to splitting the M-tree in any P2P index structure assuming that the P2P index splits peers by a ball or generalized-hyperplane partitioning. This assumption holds for currently available native and transformation indices for metric spaces. In particular, the M-Chord splits a peer by halving a one-dimensional interval to which the peer’s data is mapped. This interval corresponds to values of distance from a pivot. So, by halving the interval we define a ball partitioning. The M-CAN [12] proceeds similarly. The GHT∗ and VPT∗ [3] split data directly by a generalized-hyperplane and ball partitioning, respectively.

To split an instance of M-tree, we define a general algorithm which traverses the tree from its root to leaves and processes each object based on the decision to which partition the object belongs. The pseudo-code of this algorithm is in Algorithm 1. The first parameter is a tree node, initially set to the tree’s root. The second parameter \( \text{match} \) specifies the partitioning. The \( \text{match} \) function returns an integer value which corresponds to the partition identification. For a ball partitioning defined by a pivot \( p \) and a medium distance \( d_m \), we have \( \text{match}_{BP} \). For a generalized-hyperplane partitioning defined by a pair of pivots \( p_1 \) and \( p_2 \), we have \( \text{match}_{GHP} \).

\[
\text{match}_{BP}(a) = \begin{cases} 
1 & \text{if } d(a, p) \leq d_m \\
2 & \text{otherwise} 
\end{cases}
\]

\[
\text{match}_{BP}(R(q, r^c)) = \begin{cases} 
1 & \text{if } d(q, p) \leq d_m - r^c \\
2 & \text{if } d(q, p) \geq d_m + r^c \\
0 & \text{otherwise} 
\end{cases}
\]

\[
\text{match}_{GHP}(a) = \begin{cases} 
1 & \text{if } d(a, p_1) \leq d(a, p_2) \\
2 & \text{otherwise} 
\end{cases}
\]

\[
\text{match}_{GHP}(R(q, r^c)) = \begin{cases} 
1 & \text{if } d(q, p_2) - d(q, p_1) \geq 2r^c \\
2 & \text{if } d(q, p_1) - d(q, p_2) \geq 2r^c \\
0 & \text{otherwise} 
\end{cases}
\]

Table 1. Definitions of the \( \text{match} \) function for a ball and generalized-hyperplane partitioning.
and the objects of Part matching objects one by one. The individual cases of the this ball region instead of traversing the subtree and delet- forwardly detach the whole subtree which corresponds to is completely contained in the partition $2$.

#### 3.2. DeleteInsert algorithm

We do not apply such optimization in this paper. New trees can be optimized by a bulk-loading algorithm, but the corresponding tree. Intuitively, the construction of the procedures. The forth is more so-

#### 3.3. CloneDelete algorithm

In the M-tree, distances to parent pivots are stored. We exploit these distances by defining corresponding lower and upper bounds on distances $d(o, p)$, which allows to prune objects without actually computing the distance $d(o, p)$ [16]. In the rest of this section, we define four specific split algorithms. The first three use the general structure of Algorithm 1 and differ in the specification of the $P rocess_1$ and $P rocess_2$ procedures. The forth is more so-

#### 3.4. LeafBall algorithm

This advanced algorithm operates on leaf-level nodes represented by ball regions. The algorithm separates the regions into two sets respecting the given partitioning. If a region $R(q, r^c)$ intersects both the partitions, the objects covered by it are separated and two new regions with the same pivot $q$ are created. For each set of ball regions, a new M-tree is built – the entries representing the leaves are all in-

The pseudo-code is available in Algorithm 2. If some leaf nodes in the resulting trees are slim (storing only one ob-
ject), these objects are reinserted in the regular way and the empty nodes are deleted. This optimization is not included in the pseudo-code for legibility reasons. The LeafBall al-

3.3. CloneDelete algorithm

In the M-tree, the insertion is an expensive operation because of lots of distance computations performed. This algo-
rithm avoids an insert procedure by creating a clone of the original tree and deleting the respective objects from the trees. The action $P rocess_1$ deletes objects from the clone, while $P rocess_2$ deletes objects from the original tree. Again both the delete operations are optimized on the level of ball regions, so the whole subtree can be deleted at once. This process is expensive from the memory point of view, since it requires the double amount of memory.

4. Performance evaluation

In order to verify properties of the proposed algorithms, we use a real-life dataset consisting of ten million images

---

**Algorithm 1** DivideAndProcess Algorithm

```plaintext
Input: node, match, $P rocess_1$, $P rocess_2$
1: if node is a leaf node then
2: for all objects $o$ in node do
3: if match($o$) = 1 then
4: $P rocess_1$({$o$})
5: else
6: $P rocess_2$({$o$})
7: end if
8: end for
9: else
10: for all node entries [$q, r^c$, subtree] in node do
11: if match($R(q, r^c)$) = 1 then
12: $P rocess_1$({all objects stored in subtree})
13: else if match($R(q, r^c)$) = 2 then
14: $P rocess_2$({all objects stored in subtree})
15: else if match($R(q, r^c)$) = 0 then
16: DivideAndProcess(subtree, match, $P rocess_1$, $P rocess_2$)
17: end if
18: end for
19: end if
```

The delete operation as a reverse operation to insertion of an object, i.e., we locate the leaf node which contains the object to delete, delete the object and update the covering radii on the path to the root if necessary. We do not handle node under-flows by merging with neighboring nodes as in [26] but we issue reinsertion of all objects in the under-flown node and delete this node. Due to the application of the $match$ function to ball regions, the delete operation is per-
formed on whole subtrees, if all objects in the subtree are to be deleted. However, the insertion of the deleted objects into the new M-tree is done one by one.
Algorithm 2 LeafBall Algorithm

Input: node, match
1: \( S_1 = S_2 = \emptyset \)
2: for all leaf nodes \( N \) in the tree rooted at node do
3: get the covering region \( R(q, r^c) \) of \( N \)
4: if \( \text{match}(R(q, r^c)) = 1 \) then
5: add \( N \) to \( S_1 \)
6: else if \( \text{match}(R(q, r^c)) = 2 \) then
7: add \( N \) to \( S_2 \)
8: else if \( \text{match}(R(q, r^c)) = 0 \) then
9: create new nodes \( N_1, N_2 \)
10: for all objects \( o \) in \( N \) do
11: add \( o \) to \( N_i \), where \( i = \text{match}(o) \)
12: end for
13: establish covering regions \( R(q, r^c_1), R(q, r^c_2) \) for \( N_1, N_2 \)
14: add \( N_1 \) to \( S_1 \) and \( N_2 \) to \( S_2 \)
15: end if
16: end for
17: \{create \( M\text{-tree}_1 \) and \( M\text{-tree}_2 \}\}
18: for each set \( S \) in \( \{S_1, S_2\} \) do
19: create an empty \( M\text{-tree} \)
20: insert all leaves in the tree while having node splits disabled
21: while a node is overflowing do
22: split the node by a common node-split procedure
23: end while
24: end for

gathered from the Flickr Photo Sharing System. From these images, five MPEG-7 descriptors were extracted and the images are represented as the five vectors having together 280 dimensions. In particular, the descriptors are the color structure (CS), color layout (CL), scalable color (SC), edge histogram (EH), and homogeneous texture (HT). CS, CL, and SC reflect color characteristics of the image. Local density of edge elements and their directions (sometimes called the structure or layout) is represented by EH and acts as a simple and robust substitution for shapes. Finally, HT is a texture descriptor. The distance function which measures dissimilarity of two images, is represented by a weighted sum of dissimilarity measures defined for individual MPEG-7 descriptors. The weights are 3, 2, 2, 4 and 0.5 for CS, CL, SC, EH, and HT, respectively. The distance measure for CS and SC is the simple \( L_1 \) metric, while the measures for the others are tailored to their specific nature. The distance density of the 10-million dataset is depicted in Figure 3.

Firstly, we compare the individual algorithms on 100,000 images organized in a single \( M\text{-tree} \). In order to make the comparison objective, we used three different capacities of internal and leaf nodes. The first configuration (denoted in the tables as \( \frac{1}{2} \times \text{cap.} \)) allows 21 and 84 objects stored in internal and leaf nodes, respectively. The resulting tree had 1,951 leaf nodes. In the second configuration (\( \times \text{cap.} \)), we used 42 and 168 objects, which resulted in 1,168 leaf nodes. The parameters for the third configuration (\( 2 \times \text{cap.} \)) are 84 and 336 objects having resulted in 578 leaf nodes. We used five different pairs of pivots for the generalized-hyperplane partitioning to split the \( M\text{-tree} \).
In Table 2, there are results in “number of computations of the distance function $d$” and real time averaged over five runs. The same experiment was conducted for the ball partitioning where five different pivots with appropriate values of radius were selected. These results are available in Table 3. All settings of partitioning produced almost equally-sized partitions. The experiments were executed on a regular PC (Athlon64 2.6GHz, 2GB RAM). From the results, we read that DoubleInsert is by far slower than the others. In particular, LeafBall was 19 and 32 times faster in distance computations and 17 and 25 times in real time for the middle capacities. In all algorithms, the content of the original tree was separated by calling the match function. Nonetheless, neither of the optimizations (precomputed distances to parent pivots and match defined on regions) helped this function to avoid many distance evaluations. So, all algorithms needed almost 200,000 and 100,000 distance computations for separating objects in the case of the generalized hyperplane and ball partitioning, respectively.

By subtracting these numbers from the total distance computations, we get 100 times performance gain comparing LeafBall and DoubleInsert. The trends when the capacity of nodes is changed are interesting. In the case of the insertion-based algorithms (DoubleInsert and DeleteInsert), the split costs correspond proportionally to the node capacity. Such behavior is induced by the split algorithm of M-tree nodes which computes a spanning tree internally. This algorithm requires more distance computations if the node capacity increases. On the other hand, the costs of CloneDelete and LeafBall are inversely proportional to the node capacity. The smaller node capacity implies more leaf nodes, so these two algorithms must inspect more leaf nodes and also more internal nodes. From all the results, we can also induce that the insert operation is very expensive due to frequent node splits. The costs of insertion can be decreased by applying a bulk-loading algorithm. We have not applied any bulk-loading technique, however, the authors of [7] claim a 25% gain in CPU costs only. The other algorithm [26] to bulk-load M-tree is not studied from the CPU costs point of view, but it decreases the I/O costs by 50%.

Secondly, we tested the efficiency of resulting trees in terms of querying. We run 20 range and 20 $k$NN queries on the resulting trees $M$-tree$_1$ and $M$-tree$_2$ which both have the capacity of internal and leaf nodes set to 42 and 168 objects, respectively. We fixed the radius to 0.8 and the number of neighbors to 20 for all the queries. The range queries returned 20 objects on average. Table 4 contains average values of distance computations and real time over the batch of queries. The search performance of the trees is almost identical. Then, we inserted additional 100,000 objects respecting the partitioning which produced the trees $M$-tree$_1$ and $M$-tree$_2$. The insertion costs are included in the table and they support the claim that the insertion is expensive, because 32 distance computations were needed to insert one object, on average. After populating the trees, we repeated the same querying. Obviously, the response times increased because the content of each tree was doubled, but the efficiency of trees was retained without any regards to the algorithm used to construct the tree. We experienced the same behavior when the ball partitioning was applied and for the limited space we did not include them.

Finally, we incorporated the LeafBall algorithm in the M-Chord and built a P2P network on 10 million images. The M-Chord used twenty pivots for the transformation to the one-dimensional domain and the same twenty pivots were exploited in additional filtering in the M-tree (extension taken from the PM-tree [27]). The node capacities of M-tree were the same as above. All M-Chord peers together needed 200 million distance computations to compute the transformation and we do not include these costs in comparison. First, we built the network with the DoubleInsert algorithm for splitting the peers. In total, peer split procedure was called 499 times, so the network consisted of 500 peers. The overall costs of inserting all objects, including the split costs, was 849,822,669 distance computations. Next, we replaced the split algorithm with LeafBall and the overall cost dropped down to 611,667,738 distance computations, which is an improvement by 28%. However, these costs include insertions of objects one by one, thus, the step-wise process of building each M-tree, which needed about 603 million distance computations. The remaining portion is the costs induced by splits resulting in 28 times performance boost of the LeafBall algorithm. Consequently, one M-tree split took 20 s on average, which was reduced to 1.2 s by the LeafBall algorithm. This is a very important gain because the peer is blocked during the split, so that it cannot accept new data nor answer queries.

5. Conclusions and future work

In this paper, we have focused on algorithms to efficiently split data stored in metric-based index structures of the M-tree family. The work was primarily motivated by peer splits in structured peer-to-peer networks. We presented four algorithms and compared them on a real-life image database. The high efficiency of the proposed LeafBall algorithm was confirmed by a large-scale experiment, where we built the M-Chord peer-to-peer index structure on 10 million images. The LeafBall algorithm compared to the baseline DoubleInsert algorithm provides about 28 times performance boost.

As the future work, we want to study the possibility that the M-tree itself would find a suitable partitioning. The partitioning is then reported to the peer instead of the current procedures when the peer dictates the partitioning to the M-tree. This may bring further improvement since the M-tree
can exploit its internal structures to optimize the whole process. Another research challenge is to generalize the algorithms to partitioning principles which produce more than two partitions, e.g. the Voronoi-like partitioning.

5.1. Acknowledgements

This work was partially supported by: the EU IST FP6 project 045128 (SAPIR), the national research project 1ET100300419, the Czech Science Foundation projects 201/07/P240 and 201/06/1338. We are grateful to our SAPIR partners from ISTI-CNR, Pisa, who crawled the image dataset and extracted the MPEG-7 features.

References


Table 4. Search performance and insertion costs of the proposed algorithms in the case of generalized-hyperplane partitioning.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Dist. comp.</th>
<th>Dist. comp.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Real time</td>
<td>Real time</td>
</tr>
<tr>
<td>DoubleInsert</td>
<td>M-tree <em>1</em></td>
<td>M-tree <em>2</em></td>
</tr>
<tr>
<td>Range search #1</td>
<td>24,433</td>
<td>23,739</td>
</tr>
<tr>
<td></td>
<td>0.59 s</td>
<td>0.58 s</td>
</tr>
<tr>
<td>kNN search #1</td>
<td>39,612</td>
<td>42,772</td>
</tr>
<tr>
<td></td>
<td>0.95 s</td>
<td>1.00 s</td>
</tr>
<tr>
<td>Insert</td>
<td>3,253,968</td>
<td>3,274,143</td>
</tr>
<tr>
<td></td>
<td>77.12 s</td>
<td>76.14 s</td>
</tr>
<tr>
<td>Range search #2</td>
<td>46,680</td>
<td>44,399</td>
</tr>
<tr>
<td></td>
<td>1.14 s</td>
<td>1.09 s</td>
</tr>
<tr>
<td>kNN search #2</td>
<td>75,448</td>
<td>81,581</td>
</tr>
<tr>
<td></td>
<td>1.81 s</td>
<td>2.11 s</td>
</tr>
<tr>
<td>DeleteInsert</td>
<td>M-tree <em>1</em></td>
<td>M-tree <em>2</em></td>
</tr>
<tr>
<td>Range search #1</td>
<td>24,236</td>
<td>24,474</td>
</tr>
<tr>
<td></td>
<td>0.58 s</td>
<td>0.60 s</td>
</tr>
<tr>
<td>kNN search #1</td>
<td>39,603</td>
<td>42,795</td>
</tr>
<tr>
<td></td>
<td>0.95 s</td>
<td>1.01 s</td>
</tr>
<tr>
<td>Insert</td>
<td>2,927,322</td>
<td>2,911,228</td>
</tr>
<tr>
<td></td>
<td>69.10 s</td>
<td>77.68 s</td>
</tr>
<tr>
<td>Range search #2</td>
<td>46,489</td>
<td>44,956</td>
</tr>
<tr>
<td></td>
<td>1.13 s</td>
<td>1.10 s</td>
</tr>
<tr>
<td>kNN search #2</td>
<td>75,531</td>
<td>81,542</td>
</tr>
<tr>
<td></td>
<td>1.83 s</td>
<td>1.98 s</td>
</tr>
<tr>
<td>CloneDelete</td>
<td>M-tree <em>1</em></td>
<td>M-tree <em>2</em></td>
</tr>
<tr>
<td>Range search #1</td>
<td>24,236</td>
<td>24,148</td>
</tr>
<tr>
<td></td>
<td>0.59 s</td>
<td>0.58 s</td>
</tr>
<tr>
<td>kNN search #1</td>
<td>39,603</td>
<td>42,786</td>
</tr>
<tr>
<td></td>
<td>0.95 s</td>
<td>1.02 s</td>
</tr>
<tr>
<td>Insert</td>
<td>2,943,171</td>
<td>2,938,416</td>
</tr>
<tr>
<td></td>
<td>71.29 s</td>
<td>69.61 s</td>
</tr>
<tr>
<td>Range search #2</td>
<td>46,540</td>
<td>44,891</td>
</tr>
<tr>
<td></td>
<td>1.14 s</td>
<td>1.10 s</td>
</tr>
<tr>
<td>kNN search #2</td>
<td>75,634</td>
<td>81,527</td>
</tr>
<tr>
<td></td>
<td>1.96 s</td>
<td>1.97 s</td>
</tr>
<tr>
<td>LeafBall</td>
<td>M-tree <em>1</em></td>
<td>M-tree <em>2</em></td>
</tr>
<tr>
<td>Range search #1</td>
<td>24,213</td>
<td>24,123</td>
</tr>
<tr>
<td></td>
<td>0.59 s</td>
<td>0.58 s</td>
</tr>
<tr>
<td>kNN search #1</td>
<td>39,581</td>
<td>42,768</td>
</tr>
<tr>
<td></td>
<td>0.94 s</td>
<td>1.02 s</td>
</tr>
<tr>
<td>Insert</td>
<td>3,102,343</td>
<td>3,041,479</td>
</tr>
<tr>
<td></td>
<td>74.37 s</td>
<td>71.48 s</td>
</tr>
<tr>
<td>Range search #2</td>
<td>46,024</td>
<td>44,864</td>
</tr>
<tr>
<td></td>
<td>1.12 s</td>
<td>1.17 s</td>
</tr>
<tr>
<td>kNN search #2</td>
<td>75,321</td>
<td>81,534</td>
</tr>
<tr>
<td></td>
<td>1.79 s</td>
<td>1.94 s</td>
</tr>
</tbody>
</table>


