TOWARDS ON EXPERIMENTAL COMPARISON OF THE M-TREE INDEX STRUCTURE WITH BK-TREE AND VP-TREE

Gergő GOMBOS, János Márk SZALAI-GINDL, István DONKÓ, Attila KISS
Department of Information Systems, ELTE Eötvös Loránd University, Budapest, Hungary,
E-mail: { ggombos, szalaigindl, isti115, kiss }@inf.elte.hu

ABSTRACT

In our previous paper, we showed the M-tree index [7] using GiST in the PostgreSQL database. In this paper, we present that result and we extend that with some preliminary experimental results with other indexes. We compare the M-tree index with the BK-tree and the VP-tree indexes. These can be work in metric space with edit distance, that can be used to compare DNA sequences or melody of songs. In this paper, we compare the indexes in PostgreSQL. We use the range based queries to analyze the performance of the indexes. The result shows that the M-tree index is faster than the other two indexes.

Keywords: *Databases, Indexes, M-tree, Metric space, PostgreSQL, GiST*

1. INTRODUCTION

The most important task of the databases is to answer the queries quickly. They can achieve a low response time with indexes, which are used during the search. The most index made for the exact match and they took advantage of the sortable property of the data. If we want to search a sub-sequence and we know only just the part of the subsequence then these methods are ineffective. For example, this problem can be found in the genetic database, where we want to find some DNA sequence, or the music database, where we want to find a song based on a melody.

These problems can be solved using the metric space, where the index is based on the distance between the items. The edit distance (or Levenshtein-distance) is one of the well-known metrics which is used in metric space. This method counts the operations on an item to reach another one. This is the distance between the two items. The operations are the insertion, removal or replace one character. This solution is useful for data that have different length of item, because the removal and the insertion can change the length of the item.

In our previous work [9] we presented an M-tree [5] implementation that uses the edit distance too. In this paper, we present the implementation again and extend the paper with a comparison with other indexes, like BK-tree [3] and VP-tree [17]. These index structures also can be work in the metric space.

The source codes of the project are available at [9] .

This paper presents some related work related to metric space and index-tree structure (Section 2). In Section 3, we present the used technologies: M-tree, PostgreSQL, GiST. A picksplit algorithm is needed for the GiST, and we implemented more with different strategies. We describe these methods in Section 4. In the next section (Section 5) we present the index building time, the response time of the K nearest neighborhood and the range queries and the comparison of the VP-tree, BK-tree and the M-tree with range queries. Section 6 presents some possible use cases of the M-tree. Finally, we write our conclusions and future works in Section 7.

2. RELATED WORK

There are more results in metric space usage for data indexing. Costa et al. [6] use various metric spaces, including VP-tree in music database for melody search. They use two types of metric space: city-block and euclidean distance. In our case, we used the edit distance.

Skalak et al. [14] use the VP-tree in a music database also in their paper. They convert the songs with unquantized melodic encoding and used a simple metric for melodic comparison. In their solution, they split the song files into sequences and built the VP-tree on it. When we compare the three trees we also split the song to subsequence.

Nielsen et al. [13] use the VP-tree in computer vision to find images. They changed the VP-tree to able to use the Bregman divergences, which are efficiently on the nearest neighbor queries.

The VP-tree was originally introduced by Yianilos et al. [17]. That is built on the concept of partitioning the metric space in terms of distance thresholds from specific entries selected as so-called *"vantage-points"*. The separation results in two groups for each vantage-point, one that is closer than the threshold and the other one that contains the points that are further away. Those points that share the same category with regard to several vantage-points are likely to be close as well. Later, several variations appeared, such as multiple VP-trees [2] or VP forests [18].

Other solutions use the BK-tree [3] [1] which is specialized for distance metrics with integer values in a preferably small range (often used with edit distance) so that the nodes can be distributed into groups based on their distance to their ancestor. In our case, we use the M-tree for indexing the music data in metric space. The idea of the M-tree, as introduced by [5] and [4].

We used the PostgreSQL database to compare the indices. This database has two frameworks that help the user to create own index-tree. These frameworks are called GiST (Generalized Search Tree) [10] and SP-GiST (Space-Partitioned Generalized Search Tree) [8]. The second one provides space-partitioning capabilities.

Keyvanpour et al. [12] analyzed the various index solutions for multimedia data in an analytical view. In contrast our paper we present some experimental results. Other re-

searchers: Karsdorp et al. [11] try to solve the similarity searching problem using neural network.

3. TECHNOLOGIES USED

3.1. M-tree

The main index structure of M-tree is described in [5]. It is based on the idea of attaching a so called covering radius to the nodes in the search tree that represent the maximal distance of any of the entries in any subtree of that branch. A simple illustration can be seen on Figure 1, which displays the tree structure using arrows, shows the covering radii belonging to the nodes with red dotted lines (the radius of the largest node is marked by "R") and has some arbitrarily placed blue dots to represent the stored information. Do not be confused by the fact that this image resides in a two dimensional space, there is no absolute position for the nodes, neither for the entries, only their relative distances can be calculated.

By indexing data in this manner the opportunities to speed up queries arise by being able to make sure decisions during lookup without visiting all the leaves. For example if an entire subtree (marked by the letter "A" on the figure) fits inside the range being searched for (marked by "Q" in the figure), because its node is closed to the center of the query than the radius of the query minus the radius of the node, all of its content can surely be included in the results. On the other hand if it can be decided that an entire subtree (marked by the letter "B" on the figure) cannot satisfy a condition and thus the chance of an overlap can be ruled out, because its node is further apart from the center of the query than the sum of their radii, that branch can be completely cut off from the search. In the third case, where a subtree (marked by letter "C" on the figure) is neither completely inside, nor fully outside the search region, its children have to be further processed.

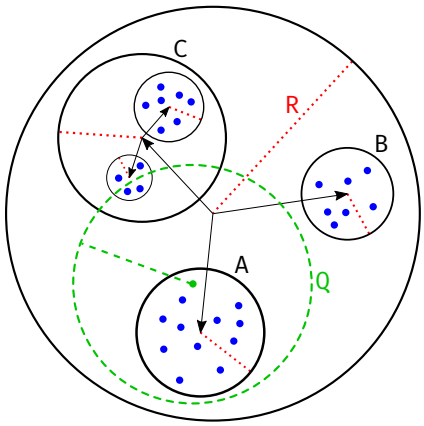

Fig. 1 Illustration of the M-tree structure, where blue dots are representing the entries storing the actual information, the red dotted lines (the longest of which is denoted by "R") are the covering radii of the nodes and the green dashed region (denoted by "Q") is an example for a range based query.

3.2. PostgreSQL

We chose PostgreSQL[1] as the database management system to give a framework because of its support for GiST (Generalized Search Tree) based index development[2] that fit the needs and constraints of the M-tree architecture. Using the `varlena` extensible struct (a variable length array type that stores its size in the first few bytes and its contents in the rest of its allocated space) as a base we defined our own datatype that includes a covering radius and the actual data in the flexible array member.

3.3. GiST

The GiST data structure is an abstract tree structure which can be considered as the generalization of other trees, such as the more well known B-tree and R-tree or in this case the M-tree. It was what permitted the extension of the system with a new index type. For that we needed to implement several functions.

The ones that belong to the index creation process were:

- `union`: Given a set of entries it creates a single node that represents all of them by selecting one of them and assigning a covering radius to it that is equal to the distance of the furthest object from the selected one and thus contain all the entries. We chose to implement it in a way that tries to select the node closest to the imaginary *center* (only intuitively, as such a center does not exists in a metric space), meaning it has the lowest possible maximal distance to every other node, so that the covering radius will be minimal.

- `picksplit`: When the count of a node's descendants reaches a certain threshold set by the database management system (usually in connection with the page size), it needs to split that node and distribute its children between the newly promoted items. This algorithm is responsible for selecting the two nodes that should be promoted in a way that ideally would later benefit the queries, for example by minimizing the overlap between the two covered areas.

- `penalty`: This method calculates how much the efficiency of the index would drop if an item were to be inserted under a certain node. This is estimated by calculating the required increase in covering radius to keep the correctness of the index. One of the advantages that [5] and [4] mention is the ability to handle dynamically changing data as opposed to other index structures which sometimes can only operate efficiently on static information. When inserting a new item the values provided by this function contribute the information needed for choosing a good path.

The others being utilized during queries:

- `consistent`: The purpose of this method is to determine whether a key that's stored in a node of the tree is *consistent* with the current query, meaning that

[1] `https://www.postgresql.org/`
[2] `https://www.postgresql.org/docs/current/gist-extensibility.html`

its subtrees can potentially contain leaves that match the specified criteria. If this test is negative, then the whole branch can be pruned from the actual search tree. There is actually also a second value returned that serves the purpose of avoiding unnecessary comparisons by allowing the implementation to tell the system when it can be determined from the observation of only one node that all the leaves belonging to its descendants must satisfy the given condition, thus do not need to be tested individually and that entire branch can be included in the results.

- `distance`: As M-trees operate on values residing in metric spaces there always exists a distance function that can determine how far two values are from one another. By providing this function we allow the index structure to be used for ordering operators, which is essential for K nearest neighbor queries as that needs to find entries based on their distance to a given *center* value, thus compare their separation from the midpoint to each other.

There would have also been four optional functions that could have been implemented (and can potentially be later added if needed) that were not necessary for our use case:

- `same`: If the representation of the data would not be injective with regards to its actual contents then this function could serve the purpose of defining when two entries should be considered equal.
- `compress`: In cases where it would be desirable to lower the amount of storage being used by the index structure, the internal tree nodes could use a smaller type than the actual data by implementing a compression algorithm in this method.
- `decompress`: When compression is applied to the stored data this function would provide the means of reconstructing the information that is needed by the other methods. (That is in case of internal nodes not necessarily the full original data.)
- `fetch`: In contrast to decompress this function would be responsible for fetching the lossless information where it is required that was initially compressed.

4. PICKSPILT STRATEGIES

We obtained test data in the form of ABC music nontation[3] (which seemed fitting for the task because of its simplicity) from a compilation made by professor José Fernando Oliveira[4] from the University of Porto. His collection can be found here: `http://www.atrilcoral.com/Partituras_ABC/index_abc.htm` This data was stripped down into a much simplified representation by only taking the notes and disregarding all rhythm related information.

The set contained 19296 entries which we shuffled to avoid organizational grouping bias and then generated different sized test sets from. (To be exact, with 1000, 3333, 6666, 10000, 13333, 16666 and the last one with all the 19296 entries.) We then built the index on these datasets using different heuristics for the picksplit algorithm and measured their performance on range based and K nearest neighbor queries. After repeating this randomization and measurement process (the data points around which the queries were placed were randomly selected from the set as well) several times we averaged the results and compared the different strategies to each other and the sequential scan as a baseline.

The testing was conducted on a machine equipped with an Intel® Core™ i5-650 CPU and 4GB of RAM running Ubuntu 16.04.

As it has been already explained above, the goal of a picksplit algorithm is to chose two representants from the group of nodes to be split, among which when the rest are distributed, queries could be performed efficiently.

The eight competing versions, including three deterministic, one completely random and finally four sampling type of strategies were:

- `PicksplitFirstTwo` - Simply just choosing the first two entries in the list to be promoted.
- `PicksplitMaxDistanceFromFirst` - Searching through the list of entries and choosing the first one and the one that is furthest apart from it in hopes of creating as little overlap as possible.
- `PicksplitMaxDistancePair` - Iterating on the previous version this searches for the two nodes in the list that have the highest possible distance out of all the combinations.
- `PicksplitRandom` - Choosing two entries randomly from the list.
- `PicksplitSamplingMinCoveringMax` - This strategy randomly picks two nodes, calculates the split that would occur if those two were to be promoted and stores the bigger covering radius of the two. After that it repeats the procedure several times (in our implementation that is a 100 times) by taking another two nodes and overwriting the results if those produce a smaller maximal radius.
- `PicksplitSamplingMinCoveringSum` - Takes samples similarly to the previously described method, but instead of trying to minimize the bigger radius of the two, it aims to find the lowest possible sum of the radii.
- `PicksplitSamplingMinOverlapArea` - This strategy calculates the imaginary overlap between the two potential regions based on their radii and the distance between them and aims to find a pair that produces a minimal intersecting area.
- `PicksplitSamplingMinAreaSum` - As a variation of the `PicksplitSamplingMinCoveringSum` strategy, instead of simply just adding together the radii,

[3] `http://abcnotation.com/`

[4] `https://sigarra.up.pt/feup/en/func_geral.formview?P_CODIGO=209980`

this method calculates some sort of a collective *area* (as if the nodes where circles in two dimensions) defined by the sum of their squared radii.

As an example, Figure 2 illustrates a situation in which two strategies would prefer different options. If presented with the possibility of splitting the node marked with A in the way shown labeled by B or C, the `PicksplitSamplingMinCoveringSum` strategy would favor B, as adding the two resulting radii together would result in a lower value there, while `PicksplitSamplingMinCoveringMax` would rather choose C, because the bigger radius of the two is smaller in that case.

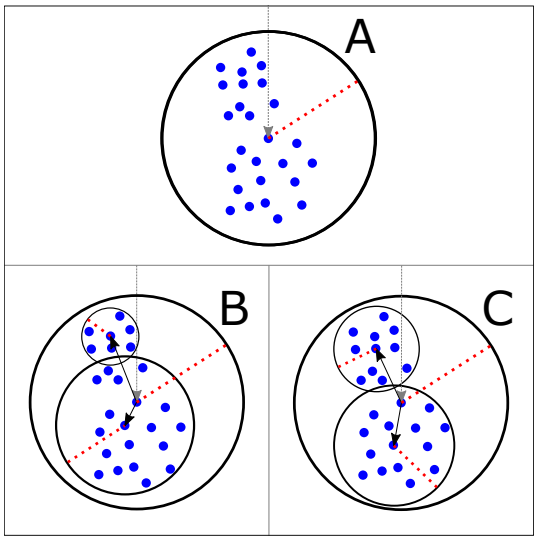

Fig. 2 Illustration of two possible options when distributing the contents of a node during a picksplit operation that are favored in two different heuristics.

5. RESULTS

When comparing these different possible heuristics for splitting we found that there was a visible trade-off between the time it takes to build an index and the query performance afterwards. Increasing the complexity of the algorithm responsible for selecting the nodes to promote generally resulted in a structure that was better suited for range based queries. There was no significant difference when comparing the K nearest neighbor performance, all the strategies far outperformed the sequential scan and were relatively close to each other.

As expected, the sequential scan performed linearly in terms of the amount of data in both tests, as it always had to traverse the entire database in order to provide the queried results.

5.1. Index building times

One of the drawbacks of our approach are the slow initial index building times, which are obviously nonexistent when doing sequential scans, but this is eased by the fact that one of the advantages of the nature of this M-tree implementation related to other spatial index structures (as it was already explained a bit in section 3) is that they are more suitable for data that is expected to change. There are approaches that can only work on static data (thus requiring a rebuild of the index whenever the underlying information changes), but in contrast to those the M-tree can also perform efficient insertions afterwards.

Usually the heuristic that minimizes the sum of the two covering radii (`PicksplitSamplingMinCoveringSum`) when splitting took the most time to build. Taking a look into it while it processed the data what became clear was that it tends to produce deeper trees, because it favours uneven splits between the two promoted nodes, as one of the covering radii being small and the other one bigger seems to result in an overall lower sum than when the two are more close to being equal.

Most likely a similar reasoning explains what happens with `PicksplitSamplingMinOverlapArea`, since it can achieve zero overlap by setting one of the radii to zero, and this results in a heavily leaning tree.

The `PicksplitRandom` strategy seems to have the fastest index building time followed by the deterministic algorithms, while the sampling type of heuristics are the slower ones, as those have to repeat the same calculations over and over again.

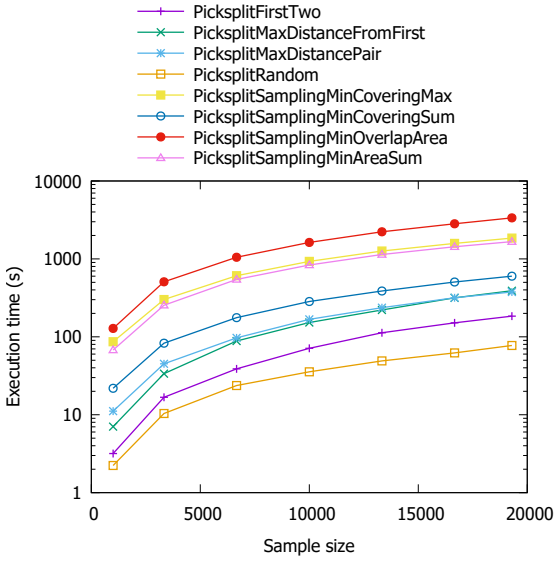

Fig. 3 The time it took to build the index with different splitting strategies for several data sample sizes.

5.2. Range based queries

The execution times for range based queries can be observed on Figure 4.

This first type of query that the M-tree supports is one where we would like to access the neighbors of a given entry within a certain radius. The execution times where greatly reduced for smaller radii (a), as there were fewer overlaps and greater sections of the tree could be avoided while searching. As expected the performance benefits start to lessen when the radius is increased (b), because fewer branches can be pruned.

As it was already mentioned in the previous section, the `PicksplitRandom` strategy took the least time to build the index, and that seems to relate to the quality of the structure, as it can be seen, it was the one that provided the smallest amount of improvement during the range based queries.

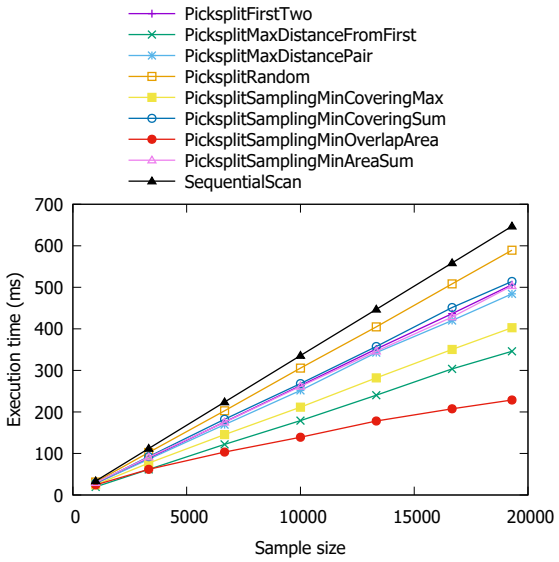

Fig. 4 The time it took to execute range based queries using indexes built with different splitting strategies for several data sample sizes. We tested smaller and larger search limits and observed that the improvements are greater in case of a shorter radius, such as the one presented here.

5.3. K nearest neighbor queries

As you can see on Figure 5, in the K nearest neighbor search the M-tree based index significantly outperformed the sequential scan.

When we first saw the results something that seemed weird was the fact that some picksplit strategies have seemingly managed to reduce their execution time when testing on a bigger data set. Our initial explanation for this was that when there are more entries, the space is more densely populated, thus the neighbors are closer together, so fewer branches have to be traversed in order to find the same amount of them, but after running multiple measurements it became more and more clear that the fluctuations in the different data sizes are more likely due to noise, as measuring such low values (around 10ms) turns out not to be accurate enough. When averaging out the results of multiple measurements the plots started to flatten into a very slowly rising line.

We also measured query execution times for exact matches, because that was the only metric in which we could possibly compare with other (non metric-only) index structures, as without defining an ordering (avoiding which is exactly the point of the M-tree) those would essentially perform equal to a sequential scan. When looking for an exact match, the M-tree was already outperformed by the sequential scan and even more so by a B-tree, but that is not surprising, as it was not designed for that use case. What we would suggest as the probable cause for that are the overlaps between the nodes with their covering radii, that forces the search to traverse multiple paths, as in other index structures overlaps are more easily avoidable. With all that said, it doesn't mean that creating an index structure with our process will reduce the performance of any queries, as the PostgreSQL query planner chooses the sequential scan over our index unless explicitly being instructed to use the index instead.

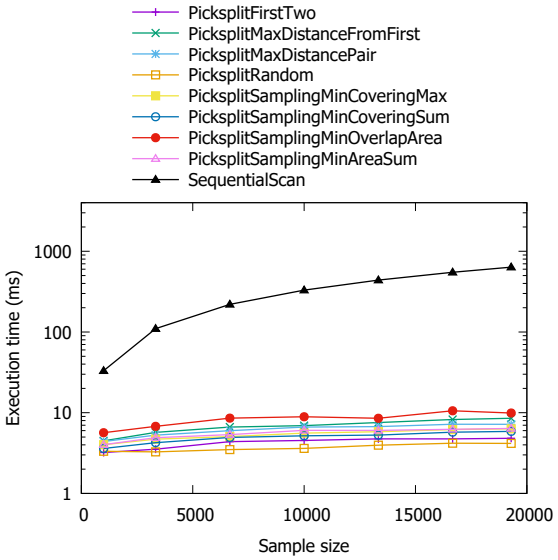

Fig. 5 The time it took to execute K nearest neighbor queries using indexes built with different splitting strategies for several data sample sizes.

5.4. Comparison of the M-tree, BK-tree and the VP-tree indexes

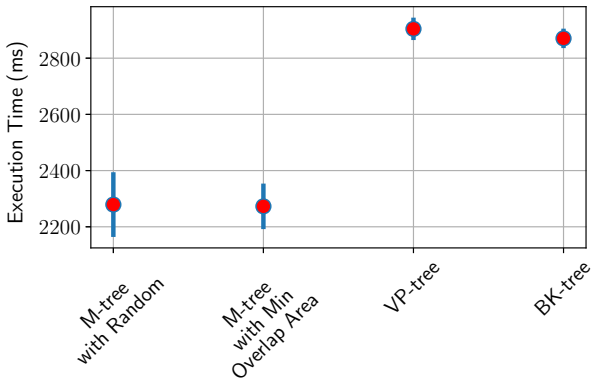

Fig. 6 The execution time of 10 range queries with distance 8. The used index structure: M-tree with PicksplitRandom , M-tree with PicksplitSamplingMinOverlapArea, VP-tree and BK-tree.

We compare the M-tree implementation with two other well-used index structures BK-tree and VP-tree. We did not find any official implementations but we found one on

[5] `https://github.com/fake-name/pg-spgist_hamming`

a Github[5] which work with SP-GiST in PostgreSQL. We change the distance function in the code because it used the Euclidean distance, but we need edit distance. We did not change any other parts of the code. It is possible that it is not an optimal code for the indexing. We used the previously presented ABC dataset, but we split the songs to subsequence. The length of every sequence is at most 15. The sequences are overlapping, like a sliding window. With this modification, we got 1,9 million rows. For the measurement, we used 10 different melodies from the database and we use the range queries with distance 8 for the comparison. Between the measures, we restart the database and clear the system cache. We used two Picksplit implementations in this case based on the previous result. We selected the PicksplitRandom and the PicksplitSamplingMinOverlapArea, because these two strategies are the best and worth for the queries. Figure 6. shows the execution time of the range queries. We see that the M-tree implementation can be faster with 600 ms then the VP-tree and BK-tree. We can see no significant difference between the two Picksplit strategies. This is a preliminary result and a future we would like to analyze deeper. We wanted to compare the indexes with K Nearest Neighbor queries, but now the SP-GiST does not support the index for the ordering. Both VP-tree and BK-tree used the sequence scan in this case, and it is not fair to compare with that. We hope the later version of the SP-GiST will support the ordering with index.

6. POSSIBLE USE CASES

In this section we present some possible real life scenarios in which the system could prove to be useful by explaining what difficulties occur and how this construction could contribute a solution to them.

6.1. Music

6.1.1. Audio based music search

Websites and applications, such as *SoundHound* and *Shazam* provide services that are capable of identifying song from either a short recording of the actual audio or at times even by just having the user sing or hum part of the melody. As far as we know, these are currently based on fingerprinting and other similar techniques. It often occurs that somebody does not remember a whole sequence correctly and thus may leave out notes or maybe even add extra notes out of imagination. In these cases with an appropriate distance definition the efficient K Nearest Neighbor queries made possible by this implementation could help in the identification of the song by matching it against a database that has an M-tree index.

6.1.2. Common origin research

Staying in the context of music, the other type of query could be utilized for the discovery of connections between for example folk music of different nations that are based on the same melodies but are extended with different ornaments by searching withing a certain similarity range.

6.2. Genetics

In the field of genetics matching sequences to each other is a very important procedure, as it allows the construction and examination of theories that can give way to many conclusions to be drawn. By developing a distance metric based on the probabilities of certain types of mutations [16] [6] [7] extracted from existing statistics and biological understanding of these processes an efficient lookup method could be created with this by building indexes on top of existing databases. A close example can be found in the article introducing ND-GiST [15], which is also an index structure created using GiST for the purpose of improving query performance on genetic data.

6.3. Publishing

Any form of media, where originality is a requirement is often faced with the challenge of checking for already existing similar works. Exact copies are easy to filter, but when an adversary alters the given information with the intent of making it harder to detect more robust methods are necessary. For example in the case of plagiarism detection, where some materials authenticity, be it text, music, picture or other work needs to be asserted, in relation to an existing database of entries, and an appropriate distance metric can be designed to discover similarities, an M-tree based construction is expected to perform quickly enough while resulting in a more tunable experience.

6.4. Spell checking and autocompletion suggestions

When typing plain text or source code it can be very convenient when by only typing a few letters a correct guess is presented, thus we do not need to type the entire word, just accept what was suggested. Several programs only offer words that contain an exact match to what we typed. Some other applications allow different parts of the desired string to be entered as to avoid having to type long parts that are the same between different suggestions, thus is unhelpful in deciding between them. What would be even better, but is rarely available in current software is the so called *fuzzy string search*, which is essentially based on a penalty system calculated using edit distance, thus is able to tolerate typing errors that add letters that are not present in the desired string. When the collection of words to chose from is sufficiently large enough (for example the words of an entire language), such a process could potentially be sped up by making use of a database with an M-tree index and K nearest neighbor queries for suggesting the K closest alternatives.

7. CONCLUSION AND FUTURE WORKS

The achieved results are already capable of providing significant performance improvements, but the field is far

[6] `http://www2.csudh.edu/nsturm/CHEMXL153/DNAMutationRepair.htm`
[7] `https://ghr.nlm.nih.gov/primer/mutationsanddisorders/possiblemutations`

from extensively discovered and there are still many ways in which query times could potentially be further reduced, for example:

Better tree splitting heuristics could be implemented that further optimize the resulting structure and by that improve the efficiency of the queries. In case of the *sampling* type of strategies measurements could be conducted to determine the tradeoff curve between smaller and larger sample sizes, as testing more potential pairs of promoted nodes certainly increases the index building times, but probabilistically should provide a structure that is more adherent to the rule of said strategy. Variations on certain existing strategies could be created by tuning their parameters, such as trying different powers in `PicksplitSamplingMinAreaSum`, for example cubing the radii would essentially make it into `PicksplitSamplingMinVolumeSum` or even higher dimensions. Multiple different strategies could be merged by each giving a score to every possible split and then the choice could be determined by a weighted sum of these scores. This way desirable properties could be combined (for example if one strategy incentivizes a favorable property while another penalizes something that is preferably avoided, these could be achieved together), resulting for example in a strategy that still aims to minimize overlap, but avoids zero area nodes because of being persuaded to create even splits by giving it a penalty based on, say, the squared difference of the radii. In cases where the calculation of the chosen metric is resource intensive a caching mechanics could be added to reduce the number of distance computation.

ACKNOWLEDGEMENT

Project no. ED_18-1-2019-0030 (Application domain specific highly reliable IT solutions subprogramme) has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme funding scheme. The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002). Supported by the ÚNKP-19-3 New National Excellence Program of the Ministry for Innovation and Technology.

REFERENCES

[1] R. BAEZA-YATES and G. NAVARRO. Fast approximate string matching in a dictionary. In *Proceedings. String Processing and Information Retrieval: A South American Symposium (Cat. No. 98EX207)*, pages 14–22. IEEE, 1998.

[2] T. BOZKAYA and M. OZSOYOGLU. Distance-based indexing for high-dimensional metric spaces. *SIGMOD Rec.*, 26(2):357–368, June 1997.

[3] W. A. BURKHARD and R. M. KELLER. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, April 1973.

[4] P. CIACCIA, M. PATELLA, F. RABITTI, and P. ZEZULA. Indexing metric spaces with m-tree. In *SEBD*, volume 97, pages 67–86, 1997.

[5] P. CIACCIA, M. PATELLA, and P. ZEZULA. M-tree an efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd VLDB Conference Athens, Greece, 1997*. IBM Almaden Research Center: Very Large Databases Endowment Inc., 1997.

[6] F. COSTA and F. BARBOSA. Timbre similarity search with metric data structures. *WEMIS 2009*, 2009.

[7] I. DONKÓ, J. M. SZALAI-GINDL, G. GOMBOS, and A. KISS. An implementation of the m-tree index structure for postgresql using gist. In *2019 IEEE 15th International Scientific Conference on Informatics*, page 6, Poprad, Slovakia, nov 2019. IEEE.

[8] M. Y. ELTABAKH, R. ELTARRAS, and WALID G. AREF. Space-partitioning trees in postgresql: Realization and performance. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 100–100. IEEE, 2006.

[9] G. GOMBOS, I. DONKÓ, and J. M. SZALAI-GINDL. Source code of the m-tree index, 2020. Available at `https://www.github.com/ggombos/mtree`.

[10] J. M. HELLERSTEIN, J. F. NAUGHTON, and A. PFEFFER. Generalized search trees for database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 562–573, 1995.

[11] F. KARSDORP, P. van KRANENBURG, and E. MANJAVACAS. Learning similarity metrics for melody retrieval. In *Proceedings of the 20th International Society for Music Information Retrieval Conference*, pages 478–485, 2019.

[12] M. KEYVANPOUR and N. IZADPANAH. Analytical classification of multimedia index structures by using a partitioning method-based framework. *The International Journal of Multimedia & Its Applications*, 3(1), 2011.

[13] F. NIELSEN, P. PIRO, and M. BARLAUD. Bregman vantage point trees for efficient nearest neighbor queries. In *2009 IEEE International Conference on Multimedia and Expo*, pages 878–881. IEEE, 2009.

[14] M. SKALAK, J. HAN, and B. PARDO. Speeding melody search with vantage point trees. In *ISMIR*, pages 95–100, 2008.

[15] J. M. SZALAI-GINDL, A. KISS, G. HALÁSZ, L. DOBOS, and I. CSABAI. Nd-gist: A novel method for disk-resident k-mer indexing. In *World Conference on Information Systems and Technologies*, pages 663–672. Springer, 2019.

[16] C. TOMASETTI. On the probability of random genetic mutations for various types of tumor growth. *Bulletin of mathematical biology*, 74(6):1379–1395, 2012.

[17] P. N. YIANILOS. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 311–321, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[18] P. N. YIANILOS. Excluded middle vantage point forests for nearest neighbor search. In *In DIMACS Implementation Challenge, ALENEX'99*, 1999.

Received March 10, 2020, accepted June 1, 2020

BIOGRAPHIES

Gergő Gombos was born in 1986. He graduated (MSc) in 2012 at Eötvös Lóránd University (ELTE) as Computer Science. He defended his PhD in 2018. The topic of his thesis are the Semantic Web and the distributed computing in Hadoop environment. Since 2018 he works as assistant professor at Department of Information Systems of Eötvös Lóránd University. His research interest are the computer networks, hadoop and spark environment, big data architecture and analysis and nosql databases. He attended a lot of internactional research projects in computer networks field.

János Márk Szalai-Gindl was born in 1985. János Márk Szalai-Gindl received his M.Sc. degree in Mathematics from the Budapest University of Technology and Economics. He is currently a Ph.D. candidate. He has written his dissertation on data-intensive methods for managing scientific data. His main research interests are in scientific databases and in data sciences. He participates in multiple research projects in those fields. He is an Assistant lecturer in the Department of Information System, Faculty of Informatics at the Eötvös Loránd University in Budapest. He was the supervisor of 12 BSc/MSc students who graduated as computer scientists.

István Donkó is a master's degree student of the Informatics at Eötvös Lóránd University (ELTE). He graduated (BSc) at the same university in 2018.

Attila Kiss was born in 1960. In 1985 he graduated (MSc) as a mathematician at ELTE Eötvös Loránd University, in Budapest, Hungary. He defended his PhD in the field of database theory in 1991; his thesis title was Dependencies of Relational Databases. His scientific research is focusing on database theory and practice, security, semantic web, big data, data mining, artificial intelligence and bioinformatics. He has more than 140 scientific publications. He was the supervisor of 7 students who received PhD. He also supervised about 130 BSc/MSc theses. He was the project leader of more than 25 industrial and research projects. Since 2010 he has been working as the head of Department of Information Systems at ELTE Eötvös Loránd University.