Query Optimization in NoSQL Databases Using an Enhanced Localized R-tree Index

Aristeidis Karras[1](✉), Christos Karras[1], Dimitrios Samoladas[1], Konstantinos C. Giotopoulos[2], and Spyros Sioutas[1]

[1] Computer Engineering and Informatics Department, University of Patras, Patras, Greece
{akarras,c.karras,samoladas,sioutas}@ceid.upatras.gr

[2] Department of Management Science and Technology, University of Patras, Patras, Greece
kgiotop@upatras.gr

Abstract. Query optimization is a crucial process across data mining and big data analytics. As the size of the data in the modern applications is increasing due to various sources, types and multi-modal records across databases, there is an urge to optimize lookup and search operations. Therefore, indexes can be utilized to solve the matter of rapid data growth as they enhance the performance of the database and subsequently the cloud server where it is stored. In this paper an index on spatial data, i.e. coordinates on the plane or on the map is presented. This index is be based on the R-Tree which is suitable for spatial data and is distributed so that it can scale and adapt to massive amounts of data without losing its performance. The results of the proposed method are encouraging across all experiments and future directions of this work include experiments on skewed data.

Keywords: Big data · NoSQL · Indexes · R-tree · Range queries · kNN

1 Introduction

Indexes are data structures that originated from the urge to rapidly locate data contained within databases. In order to search for all records that belong to a specific range or to generally fulfil certain criteria, users have to manually retrieve each item and determine if it meets the parameters entered. For a database with N items, this would need $\mathcal{O}(N)$ time, which is impractical for the massive databases existing nowadays due to Big Data era. Consequently, an index is any data structure that helps speed up the search process. However, to perform so, additional storage writes and storage space are utilized. There are a number of indexes that meet the varying requirements of geographical, chronological, textual, and multidimensional data, among others. Choosing the appropriate index for a specific use-case is a crucial aspect of the whole procedure since it might lead to time complications while searching operations can vary between $\mathcal{O}(logN)$ and $\mathcal{O}(1)$ time.

2 Related Work and Motivation

The main motivation behind this work is smart query optimizers as the ones presented in [1,6,9–11,13] along with indexing schemes. Moreover, optimized versions of R-tree structures are presented in [3,4] whereas R-tree is used along with machine learning methods to create a learned index which mainly focuses on instance-optimized components. LSM-trees [12] are also of note, while their optimized versions are presented in [2]. Moreover, an LSM index for spatial data is shown in [5]. Spatial analytics require models that are hybrid structures as in [8] utilizing R+ tree. Finally, multidimensional indexes are presented in [7,14].

3 Methodology

Having briefly discussed the efficient query optimizers, our optimized index is presented here which is constructed using the widely-used R-Tree structure.

The distributed implementation comprises the procedure for constructing the distributed R tree, followed by the capacity to conduct searches based on range queries and nearest neighbours. The tree which is produced and stored in HBase is static, meaning that no further input is permitted after its creation. To construct the tree, a MapReduce task is utilized which involves a mapper and a reducer function. The technique here is to separate the dataset and generate local R-Trees for each element and then the trees are then stored within HBase.

The internal structure of an R-Tree record in an HBase database consists of a `ROWKEY`, the `MBR` of the node[1]. The children of the node are kept in a family of columns entitled children and each child corresponds to a column entitled child and an index numbers the children. For the leaf nodes, the `child0` column always includes the word leaf so that it can be retrieved while traversing the tree, that a leaf is reached. In the proposed method, the data contained in the dataset is saved inside the records. HBase can handle extremely broad rows, therefore it is useful to be able to get all of the entries within a sheet with a single visit to the database rather than sending many `GET` calls. Obviously, this holds true if the data corresponding to the points in the dataset is not enormous and the values in the sheets include the `ROWKEY` of a record from the table storing the data.

In order to traverse the tree it is enough to use the values stored in the columns as `ROWKEY` in subsequent functions of `GETs` operations. The root in the traditional sense does not exist but there is an external file containing the `ROWKEYS` (which are a list of `MBRs`) of its children. To store the R-Trees created by each reducer in HBase, we use two models:

– **A Localized Index**: After partitioning the dataset, the local R-Trees are stored in different HBase tables and the information to access them for searches is stored in a file on the local system.
– **A Global Index**: After partitioning the dataset, the local R-Trees are all stored in an HBase table whereas the information is stored in a local file.

[1] which stores the coordinates in ascending order, beginning with the lower left corner of the rectangle and ending with the top left corner.

3.1 Enhanced Range Query for Localized Index

The range query search is based on the initial algorithmic structure whereas the only difference is that we do not have a root in the R-tree but we instead read the `MBRs` from an index file and check whether there is an element that intersects our search area. This improved version is shown in Algorithm 1. This version uses parallelization among threads to simultaneously search different HBase tables containing local R-Trees. Hence, the search space in the query spans 2 children of the root and thus when used in two tables, the search is optimized. A synchronized method is used for results printing, which prevents it from being executed simultaneously by more than one thread. Moreover, all threads use a single connection to HBase for scalability. Therefore, if a large number of users try to query the index, the system can cope with 1 connection/user. If the same search is performed simultaneously by many users then the load increases significantly but in the Localized Index each thread operates on a different table so the search load is evenly distributed among the tables in the worst case[2].

Algorithm 1. Enhanced Range Query for Localized Index

```
1: procedure range search(search area)
2: $connection \longleftarrow$ connectToHBase()
3: $file \longleftarrow$ read file (index file)
4: for each line in file do
5:     $table, mbr \longleftarrow line$.split()
6:     if $mbr$ intersects $SearchArea$ then
7:         startThread(SearchFunction($SearchArea, table, connection$))
8:                                         ▷ Default range search algorithm
9:     end if
10: end for
11: waitThreads()
12: end procedure
```

4 Experimental Results

In this Section, indexing, range queries, and k-nearest neighbour search performance is evaluated. Three synthetic datasets of 5,000,000, 12,000,000, and 24,000,000 entries are utilised in the experiments. Each consists of a two-dimensional point in Euclidean space and a label. The datasets were created by a Python script that randomly generates user names from a $[-10000, 10000]$ uniformly[3] and separates the lines based on first and last names.

[2] Even when many users execute the same query at the same time.

[3] This indicates that there are no thick or sparse regions within the space covered within the datasets and ensures that the burden is dispersed evenly across the reducers.

4.1 Index Construction

To construct the index, two models and a centralised implementation are compared in terms of time for each of the datasets as shown in Fig. 1. The maximum number of children per node $\mathcal{M}$ is set to 100 in all experiments, whereas the number of reducers is set to 6. Note that it was essential to increase the Java memory capacity to at least 4 GBs in order to generate the index in the centralised approach for datasets containing between 12 and 24 million records.

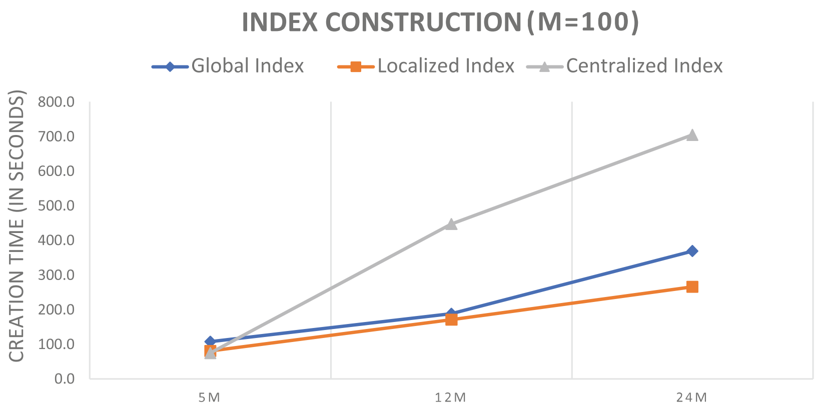

Fig. 1. Indexing functions for Centralized, Global and Localized implementation.

The experiments show that both of the developed models have excellent scalability as the index construction time rises in compact-size as the data size grows[4] compared to the centralised approach, where the duration varies from around 1 min for 5 million records to almost 8 min for 12 million sorted[5] records. In terms of generation speed, the Localized model seems to outperform the Global model as the amount of data grows.

4.2 Range Queries

The range search are compared among the two implemented models and the MapReduce implementation as shown in Fig. 2. In order to evaluate the performance of the different approaches, a comparison is done using range queries on a dataset containing 12 million records and spanning a square-shaped region in the center of the space. For the Localized implementation, six reducers were used, resulting in six partitions along the x axis and six HBase tables. Four queries were executed for each technique with square sides and each query was executed four times. We take the average search time as the evaluation metric.

[4] The distributed index contains Terabytes of records.

[5] Because sorting is being done by HBase, `PUT` functions on a lower table size are substantially quicker than on a much bigger table.

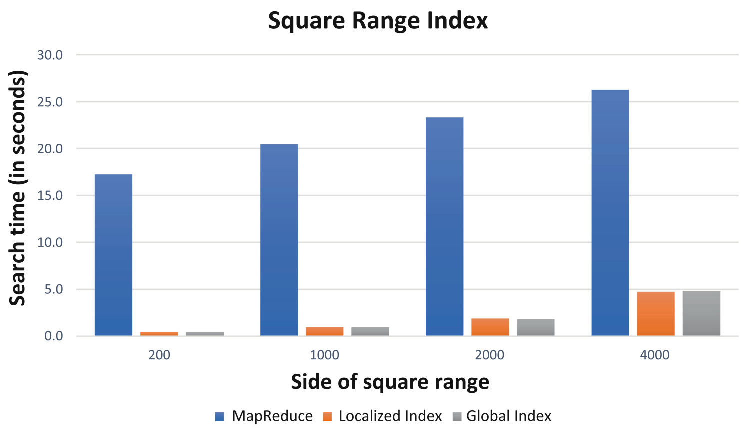

Fig. 2. Square range search times for MapReduce, Localized and Global Indexes.

As we can observe, the MapReduce search solution is unsuitable for tiny datasets and lacks in time due to the additional time necessary to launch a MapReduce job, which consumes at least 40% of the time. Hence, the implementation in MapReduce is not recommended here and is not further discussed. Regarding the two distributed index models, the implementations provide comparable performance, and it seems there is no optimal solution. However, for the Localized technique an enhanced version that parallelizes the operation by separating the index into many arrays is used. In Fig. 3, the improved approach is compared to its predecessors for the specified quadratic ranges.

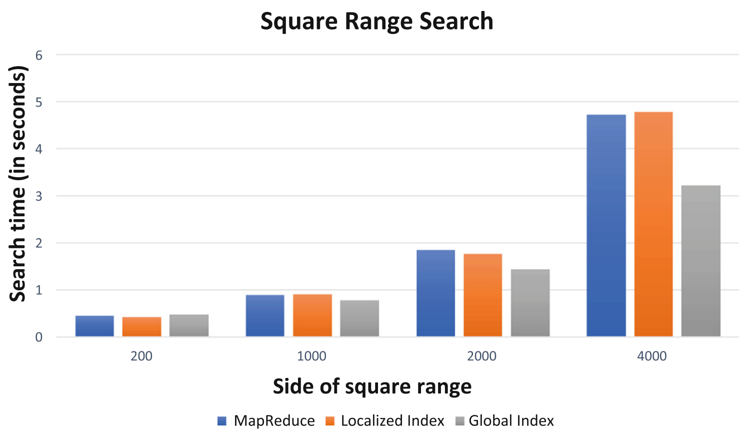

Fig. 3. Simple square range searches with an enhanced version of Localized search.

To improve the lookup version to perform optimally, ranges that cover many tables simultaneously are required. Figure. 3 shows that when the range is very small, the enhanced version shows almost no difference from simple searches, and in the case of 200, it performs lower due to the extra time spent on starting a single thread rather than performing a direct search in a specific table. Due to the fact that index partitioning into arrays is performed using the $\mathcal{X}$-coordinate, the enhanced search is most effective when dealing with long ranges. Hence, the higher the improvement, the longer the search. Due to that, the measurements were repeated with a search length such that searches are performed concurrently on all six tables of the Localized index aforementioned, while we progressively increased the search width. Increasing the search breadth results in increased thread burden as shown in Fig. 4.

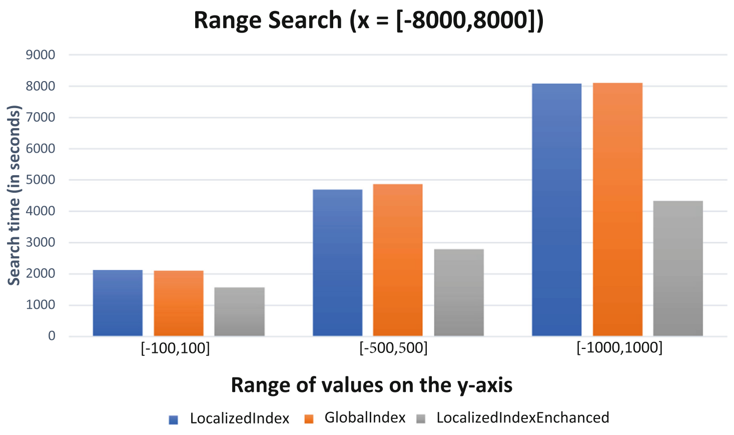

Fig. 4. Lookup times for long-length and variable-width ranges.

The improvement for long-distance searches is significantly larger. In fact, the time required for a search with a breadth in the range $[-1000, 1000]$ is about twice that of typical search techniques. Therefore, we infer that as the number of searches done by each thread rises, the search becomes faster as more searches occur concurrently and parallelism is maximised.

4.3 kNN Queries

Both index models implement the same methods for closest neighbour queries similarly to the centralised implementation. The measurements were performed once again on the dataset consisting of 12,000,000 elements and concerned a position, which we refer to as the Center Point, situated in the centre of the dataset and a point placed outside the dataset, which we refer to as the Remote Point. The searches for the two locations were conducted with progressively increasing k (number of neighbours) to determine the evolution of the two models over time in this form of search. Figure 5 (left) depicts the findings for the Center Point, whereas Fig. 5 (right) depicts the Remote Point.

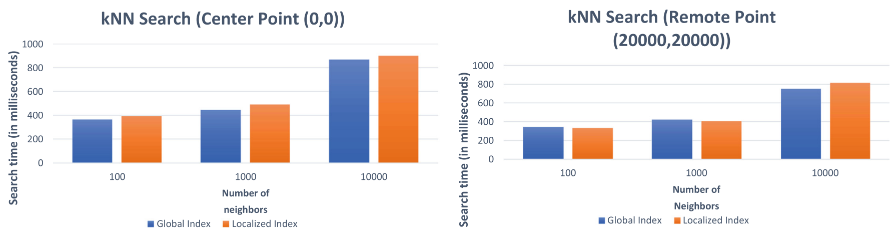

Fig. 5. Nearest neighbor search for the point (0,0) (left) and (2000,2000) (right) located in the center of the space covered by the dataset.

5 Conclusions and Future Work

In the context of this work, an enhanced localized index was created that greatly reduces the time required for different queries compared to basic brute-force searches and MapReduce. In addition, the index can manage increasing data volumes without wildly straining its construction time and performance. This is a useful feature in the era of Big Data, where data is rapidly expanding. However, the index is static, meaning that it cannot be modified after its creation, although this in no way diminishes its use. Data analytics in organisations and identifying the objectives attained by a firm over a given period of time are two examples of activities that demand rapid access to static data gathered over time. This data is hundreds of gigabytes in size and does not change over the course of the investigation. Consequently, the distributed index constructed may be a valuable tool for indexing and searching them quickly and effectively.

In addition to determining the overall utility of the index, the efficiency of the implemented index models must be determined. While both indexes in their traditional form exhibit comparable performance, the Localized index is the correct answer due to its flexibility and room for development. As we have seen, several tables enable us to take use of parallelism in the search without overwhelming a single table. In addition, it was determined that the Localized index derived from the measurements used to construct the index (Fig. 1) scales better in time as the input data of the global index rises.

Additionally, splitting the index into arrays provides an additional level of locality-based data distribution, since each array represents a vast area in space. Generalizing to non-spatial data as well, each of these tables may be a collection of linked texts, sensor readings that happened within a certain time period. This enables us to rank the index depending on the significance of each table and the amount of search traffic it gets. The aforementioned flexibility is lacking in the Global approach, which stores the whole index in a single table and distributes the data depending only on the partitioning performed by HBase in the RegionServers. In order to disseminate this table among multiple network nodes, we must first partition it, a time-consuming and error-prone procedure for large volumes of data. Additionally, a RegionServer may have acquired more popular data, resulting in an unequal distribution of load.

Future directions of this work include the handling of data skew, i.e. the scenarios in which the dataset has highly dense or very sparse portions that result in an unbalanced load distribution among the reducers.

References

1. Babcock, B., Chaudhuri, S.: Towards a robust query optimizer: a principled and practical approach. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 119–130 (2005)
2. Dayan, N., Athanassoulis, M., Idreos, S.: Optimal bloom filters and adaptive merging for LSM-trees. ACM Trans. Database Syst (TODS) **43**(4), 1–48 (2018)
3. Gu, T., Feng, K., Cong, G., Long, C., Wang, Z., Wang, S.: A Reinforcement Learning Based R-Tree for Spatial Data Indexing in Dynamic Environments. arXiv preprint arXiv:2103.04541 (2021)
4. Haider, C., Wang, J., Aref, W.G.: The AI+ R-tree: An Instance-optimized R-tree. arXiv preprint arXiv:2207.00550 (2022)
5. He, J., Chen, H.: An LSM-tree index for spatial data. Algorithms **15**(4), 113 (2022)
6. Izenov, Y., Datta, A., Rusu, F., Shin, J.H.: COMPASS: Online sketch-based query optimization for in-memory databases. In: Proceedings of the 2021 International Conference on Management of Data, pp. 804–816 (2021)
7. Langendoen, K., Glasbergen, B., Daudjee, K.: NIR-Tree: A Non-Intersecting R-Tree. In: 33rd International Conference on Scientific and Statistical Database Management, pp. 157–168 (2021)
8. Liu, Y., Hao, T., Gong, X., Kong, D., Wang, J.: Research on hybrid index based on 3D multi-level adaptive grid and R+Tree. IEEE Access **9**, 146010–146022 (2021)
9. Marcus, R., Negi, P., Mao, H., Tatbul, N., Alizadeh, M., Kraska, T.: Bao: Making learned query optimization practical. ACM SIGMOD Rec. **51**(1), 6–13 (2022)
10. Marcus, R., et al.: Neo: A learned query optimizer. In: Proceeding of the VLDB Endow. **12**(11), 1705–1718 (2019). https://doi.org/10.14778/3342263.3342644
11. Markl, V., Lohman, G.M., Raman, V.: LEO: An autonomic query optimizer for DB2. IBM Syst. J. **42**(1), 98–106 (2003)
12. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (LSM-tree). Acta Informatica **33**(4), 351–385 (1996)
13. Sellami, R., Defude, B.: Complex queries optimization and evaluation over relational and NoSQL data stores in cloud environments. IEEE Trans. Big Data **4**(2), 217–230 (2017)
14. Sprenger, S., Schäfer, P., Leser, U.: BB-Tree: A main-memory index structure for multidimensional range queries. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 1566–1569 (2019)