Article

Intelligent Indexing—Boosting Performance in Database Applications by Recognizing Index Patterns

Alberto Arteta Albert¹,*,†, Nuria Gómez Blas²‡, and Luis Fernando de Mingo López²†

¹ College of Arts and Sciences, University of Troy, Troy, AL 36082, USA
² Escuela Técnica Superior de Ingeniería de Sistemas Informáticos, Universidad Politécnica de Madrid, 28031 Madrid, Spain; nuria.gomez.blas@upm.es (N.G.B.); fernando.demingo@upm.es (L.F.d.M.L.)
* Correspondence: aarteta@troy.edu; Tel.: +1-334-670-3409
† These authors contributed equally to this work.
‡ Current address: 112C McCall Hall (MSCX), Troy University, Troy, AL 36082, USA.

Received: 16 July 2020; Accepted: 18 August 2020; Published: 20 August 2020

Abstract: An issue that most databases face is the static and manual character of indexing operations. This old-fashioned way of indexing database objects is proven to affect the database performance to some degree, creating downtime and a possible impact in the performance that is usually solved by manually running index rebuild or defrag operations. Many data mining algorithms can speed up by using appropriate index structures. Choosing the proper index largely depends on the type of query that the algorithm performs against the database. The statistical analyzers embedded in the Database Management System are neither always accurate enough to automatically determine when to use an index nor to change its inner structure. This paper provides an algorithm that targets those indexes that are causing performance issues on the databases and then performs an automatic operation (defrag, recreation, or modification) that can boost the overall performance of the Database System. The effectiveness of proposed algorithm has been evaluated with several experiments developed and show that this approach consistently leads to a better resulting index configuration. The downtime of having a damaged, fragmented, or inefficient index is reduced by increasing the chances for the optimizer to be using the proper index structure.

Keywords: autoindexing; learning indexes; index patterns; intelligent databases

1. Introduction

A database is an organized collection of data, generally stored and accessed electronically from a computer system. The Database Management System (DBMS) is the software that interacts with end users, applications, and the database itself to capture and analyze the data. The DBMS software additionally encompasses the core facilities provided to administer the database. Computer scientists may classify database-management systems according to the database models that they support. Relational Databases Management Systems (RDBMS) became dominant in the 1980s. These model data as rows and columns in a series of tables, and the vast majority use Structured Query Language for writing and querying data. In the 2000s, non-relational databases became popular, referred to as NoSQL because they use different query languages.

Creating indexes is the main task to improve database query performance, since the indexes are the most used technique to speed up queries response. It is important to properly choose the columns to be indexed, given that it also affect time to insert and update data and increase disk consumption. The DBMS Optimizer is responsible for analyzing queries and choosing the most efficient way to access information [1,2].
The primary purpose of an index is to accelerate the execution of queries in database applications. Generally speaking, it does this by diminishing the quantity of database data pages to examine. While primary or clustered indexes [3] are usually organized following the physical order of data in a table [4], the secondary indexes are copies in memory of the column they refer to along with a 25 row identifier. The indexed tables are usually arranged in two forms [5].

1. Heap tables—the data is not sorted in any way, it is just a pile of unordered, unstructured records.
2. Index-sorted out tables—Indexing orders rows in a specific sequence, usually in ascending or descending order on one field. Indexing creates a list of rows arranged in a logical order, such as by date or by name, and stores this list in a separate file called an index file.

The advantages of utilizing indexes when searching are beyond doubt, even when data is used for local sorting operations in temporal tables [6,7]. However, dynamic or intelligent indexing offers many possibilities to improve the conventional indexing operations [3,8]. Currently, index recommendations and index monitoring in DBMS are made by Query Optimizers (QO). These optimizers check the internal statistics to determine when and index should be created, removed, or recreated [9]. The approach that standard optimizers use is very straightforward, leaving room to more sophisticated learning algorithms. That theoretically could be more accurate, more flexible, more proactive, and, therefore, more efficient in the automatic creation, recreation, removal, or even modification of necessary indexes [9,10]. Choosing the right index also can be challenging when running parallel queries [11] or in databases that contain image fields [12]. Static indexing reorganizes the pages necessary to index by utilizing the internal memory page numbers, which might not be the most efficient method. Dynamic indexing [13], however, is defined to predict downtimes and to reorganize its internal index structure to improve the overall performance when retrieving the desired data.

Other researchers have used Genetic Algorithms (GA) to improve database performance [14] as an automatic way to find the best set of indexes for a database. Different from our approach, their fitness function is based on time spent in insert operations and a single query, and they make experiments with just one table. Some authors modeled an index tuning architecture applied to hybrid storage environments using GA to create indexes [15]. Another model [16] tries to optimize the response time of a set of queries by creating indexes in a RDBMS. They use the response time of each group of queries to measure their performance and then compare it with the Oracle Tuning Advisor software.

1.1. Optimizers

The goal of a QO is to find options for running a given query and evaluate the cost of each choice, so that the chosen one would provide the best performance for retrieving the data. Usually, the optimizers are based on statistical estimators that take into account the computational costs, and the cardinality of rows returned from specific queries. These optimizers also count the number of column hits to consider suggesting index creation. This module is usually known as index advisors. One of the essential tasks for a QO is to determine the usage of an index. The intricate details of traditional optimizers are closely related to this volume of hits as the leading indicator for a suggestion of index creation. It is the main reason why index suggestions are not considered totally accurate, especially when an index is created, and execution plans barely ever find its use as they turn to be the highest in cost. On top of that, traditional optimizers do not consider Machine learning (ML) algorithms when looking at the statistics to either estimate the best execution plan or to advise for an index creation or removal. The view is simplistic as suggest to remove the least utilized index or to add an index in a column that is often accessed [17].

1.2. Index Types

The purpose of storing an index is to optimize speed and performance in finding relevant results for a search query. Without an index, the search engine would scan every data in the database, which would require considerable time and computing power. For example, while an index of 10,000
records can be queried within milliseconds, a sequential scan of every record could take minutes. The additional computer storage required to store the index, as well as the considerable increase in the time required for an update to take place, are traded off for the time saved during information retrieval.

Primary index: A primary index is an index on an arrangement of fields that incorporates the primary key for the area and is ensured not to contain copies. It is commonly known as the clustered index [3]. Primary indexes include the critical fields of the table and are naturally made in the database when the table is created, and a Primary key is defined. There are two types: Dense and Sparse.

- **Dense Index:** For every principal value in the data file, an index record links to it that contains the search key and also a reference to the first data record of the table, see Figure 1.

- **Sparse Index:**
  - The index record does not appear often.
  - The search begins with the history pointed by the first index, and then it proceeds along with the other pointers in the file.

Secondary index: A secondary index is an index that may have duplicate values, see Figure 2. It is mainly used for a random search.

**Figure 1.** Dense vs Spare index. Dense indices are faster in general, but sparse indices require less space and impose less maintenance for insertions and deletions.

**Figure 2.** Secondary index. The secondary Index can be generated by a field which has a unique value for each record, and it should be a candidate key. It is also known as a non-clustering index.
2. Internal Methodology

2.1. B-tree Indexing

Some DBMS uses B-tree indexes for implementing the logical indexes [18]. These consist of multiple levels and are typically conceptually represented with three of them, Figure 3.

![Figure 3](image)

*Figure 3.* A diagram showing the standard three-tier structure of a B-tree index. The root node is the top level and lists the end of a range of key values represented on each of the intermediate nodes in the middle level. There can be more than one level of intermediate nodes, with higher levels of intermediate nodes pointing to other intermediate nodes instead of leaf nodes. The intermediate nodes point to the leaf nodes. The leaf pages hold the key values along with Record IDs (RIDs) that tell Database Management System (DBMS) where to find the data on the data pages of the table.

For optimal index access, DBMS requires a specific index key, and would only have to read the targeted data. The DBMS could then use the (Record ID) on that leaf page to pull additional data from the table data page. The RIDs themselves consist of a page number that is relative to either a table (SMS table spaces) or relative to a table space (DMS table spaces) and a slot number on that page. This access plan allows the engine to access any row in a table. In Figure 1, the index is created on a single char column; when looking for the index key value of E, the DBMS would start at the root node, and then will be redirected to the node that contains D, F, and J.

2.2. B+ Tree Indexing

All records of a table in a RDBMS are placed in a quasi-circular structure to ensure their changelessness should an illegal or a wrong operation occurs. The records of the indexed table are sorted out in pages that are neither accessed by the DBMS, nor by the Operating System. When a specific query is submitted to the RDBMS, such a query is interpreted and build. Its execution returns the records in the table. The way the query is interpreted relies upon the specific stockpiling structure of the records in the document. Records are usually sorted out in a heap-form. When some request the records of its traits’ esteems, the structure of the internal pages is rearranged [17,18].

To recover records in a heap sorted out a page or document, all the RDBMS does is a successive managing output of the pages where the records set. On the off chance that the rows are in an arranged page, a binary search on the circle pages of the document is performed to recover records as long as the query’s criteria utilize the same characteristics from the ones used to keep up the arranged request of those records. For instance, if a page storing the records for a table named STUDENT is arranged on the last name and first name (in a specific order), at that point binary search can be utilized to process questions that search for understudies by last name just or inquiries that search for understudies by their full name. A query that searches for understudies by their date of birth must be prepared to utilize a successive output of the arranged record, correspondingly, as long a the query searches for understudies by their first name.
A table index is an information structure and its related algorithms that contribute to a system where the records of the table can be searched more proficiently than using either consecutive sweep or binary search. The most widely recognized index structures utilized by RDBMS are B+-trees (see Figure 4)—and Hash tables [17].

The next section shows the first attempt of a so-called hybrid algorithm, inspired by hybrid algorithms used in fuzzy systems [6]. That mixes multiple organizational units of indexes and pseudo-intelligent operations to recreate new index-based structures by merging different data structures automatically [8].

The algorithm for merging B+tree to Compact B+tree is very straightforward. The first step is to combine the new items from the dynamic stage to the leaf-node array of the Compact B+tree in the static stage, using the in-place merge sort algorithm. Then, based on the merged leaf-node array, the algorithm rebuilds the level of the internal node by level bottom-up. Masstree and Adaptive Radix Tree (ART) [19], uses recursive algorithms for the merging operations. The figure shows the pseudo-code for merging Masstree to Compact Masstree. The algorithm is a combination of merging sorted arrays and merging tries. We define three merge tasks that serve as building blocks for the merge process: merge two trie-nodes, insert an item into one of the trie nodes, and then create an additional trie node to hold two objects. Note that the “==” sign between objects in the pseudo-code means that they have equivalent key slices.

The initial task is to merge the root nodes of the two tries, as shown in the `merge_nodes(root_m, root_n)` function below. Merging any two trie nodes, including the root nodes, involves merging the sorted arrays of keys within the nodes. Conceptually, the algorithm proceeds as in a typical merge sort, except that it recursively (depth-first) creates new tasks when the child nodes (or suffixes) require further merging. The merge process ends once the root node merge completes. These merging operations are performed by using the hierarchical cellular trees [20]. Merging ART to Compact ART adopts a slightly more complicated recursive algorithm. Instead of checking the key suffixes directly within the node (as in Masstree), ART method has to load the full keys from the records and extract the suffixes based on the current trie depth. The two optimizations (lazy expansion and pass compression) in ART [9] further complicates the algorithm because child nodes of the same parent can be at different levels. However, it is essential to note that this algorithm has no use for temporary objects in the database [20].

Our proposal, with main program in Algorithm 1, adds the method `merge_arrays`, see Algorithm 2, which calls the method `merge_nodes` partially inspired by References [9,20] and merges the sorted arrays as long as the threshold is met. Although some work on intelligent indexing has been done, presently, there is no such work, important enough to show the relevance of this
contribution to the field. Virtually all the main known DBMS are not embedding sophisticated ML techniques such as Artificial Neural Networks (ANN).

**Algorithm 1 Main Program**

Train the ANN model with training datasets.

```plaintext
for each index I_x in DB do
    Test Model (I_x) // cross validating the model
    switch (ANN model (threshold1)):
        case 0: Index not recommended
        case 1: Index recommended
        case 2: Index recommended after merging:
            merge_arrays((I_x_split1), (I_x_split2)) // see Algorithm 2
    end case
end for
```

**Algorithm 2 Merge Arrays (m, n)**

```plaintext
link n to parent
merge arrays (node m, n):
    // 2 running cursors: x for m, y for n for item x in m and item y in n:
    if x = y then
        // equal keyslice recursively invoke:
        case 1: merge nodes(x.child, y.child, n) // both x and y have child:
        case 2: x has child, y has suffix: add_item(y.suffix, x.child, n)
        case 3: y has child, x has suffix: add_item(x.suffix, y.child, n)
        case 4: x.suffix != y.suffix: create_node(x.suffix, y.suffix, n)
    end if
    else
        move min(x, y) to new position in n
    end if
    add_item(item x, node n, parent): // insert x to the sorted arrays in n
    insert_one(x, n)
    link n to parent
    insert_one(item x, node n):
        if x = (any item y in n) then
            // recursively invoke:
            case 1: y has child: add_item(x.suffix, y.child, n)
            case 2: y has suffix: create_node(x.suffix, y.suffix, n)
        else
            insert x to appropriate position in n
        end if
    end if
    create_node(item x, y, node parent): // create a new node for x and y
    n = new_node(x, y)
    if x = y then
        create_node(x.suffix, y.suffix, n) // link n to parent
    end if
```
3. Case Study

For building the proposal, it has been necessary to establish a real case of study in a production environment. Below there is a sample of that environment and a description of the indexes’ behavior for a particular table. This section helps to understand the index dynamics, so the predictive model can be easier to be understood.

Let us assume to have a relationship that stores some client information. Each person (first name, last name, company, address, city, state); 10,000 contacts. Here is how a traditional index would work when looking for random searches filtering by city or name (see Figure 5).

**Figure 5.** A table linked by a traditional secondary index.

### Scenarios

At the point when the DBMS is arranging the execution of a query, there are different potential scenarios for our case of study. This work only focuses on the ones that can potentially damage the system’s performance.

1. The query optimizer decides the use of an index per joined table per question. Some indexes can exist on a given table, and the optimizer chooses one of them. Additional indexes existing in either table, may not ever be used for any query, which means that the overhead of keeping up other indexes does indeed overuse the resources. Furthermore, there might be indexes that are barely used. Just the fact they exist, makes the optimizer spend time in deciding if these are used or not by checking its internal statistics. Unessential indexes create an issue for the organizer who must choose a less proficient index versus a more effective index (see Figure 6).

2. A query uses a low cardinality index. The cardinality of an index is the number of entries inside it. A database table may have a billion of rows in it, yet if an index table only holds eight entries, the cardinality is considered low [19]. A low cardinality index is not a noteworthy proficiency pick up. Most indexes are twofold pursuit trees (B-Trees) [17]. Versus a serial output of each line in a table to discover coordinating limitations, a B-Tree logarithmically diminishes the number of examinations that must be made. The additions from executing a pursuit against a B-Tree are low when the span of the tree is little. Thus, sparing the database indexes for fields with high cardinality to guarantee the additions from filtering a B-Tree is the biggest versus successive sweeps.

3. Using multi-column indexes. Making an index on multiple columns enables the DBMS to use multiple entries when querying a simple index. Nevertheless, frequent compelling queries against multiple columns are an excellent fit for multi-column indexes. Moreover, multi-column indexes can, in any case, be used by the organizer when most of the columns in the index are not obliged upon. For instance, if a multi-column index is used in a key column, a timestamp column is either
selected or filtered by the WHERE clause. The index can be reused successfully when only the key column and timestamp column are accessed together. As long as the subset of columns follows the index definition left-to-right, that multi-column index is adequately and effectively used [13]. Thus, utilizing a similar illustration, regardless of the DBMS in place. However, it might not be a good idea to use such an index when the timestamp is a composition of begin time and stop time columns, and they are two columns of the index definition. That means that a multi-column index over a few columns could end up being excessive and redundant [17].

4. Proposed Model

The proposal consists of two parts. The first one describes the pattern detection model to be used, which is a standard ANN with a topology of 3 layers, and the data sets to be used in the training and testing phase. The second subsection shows the cases this model can operate on, and the last paragraph introduces the algorithm that utilizes the model to use (and merge) or not to use a given index.

4.1. Artificial Neural Network

ANN are non-linear bioinspired systems simulating the biological neurons. ANNs are classifiers and predictors. The in/out combination, that is, the transfer function of the network, is not programmed but obtained through a "training" process on empiric datasets. The data that feeds the ANN is gathered from the index statistics of the DBMS (Table 1).
**Table 1.** Index statistics of the DBMS. These data are used as the neural network training dataset.

<table>
<thead>
<tr>
<th>Index Type</th>
<th>Size (kb)</th>
<th>Seeks</th>
<th>Lookups</th>
<th>Scans</th>
<th>Number Updates</th>
<th>Last Seek</th>
<th>Last Scan</th>
<th>Laste Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLUST</td>
<td>17</td>
<td>81,480</td>
<td>1004</td>
<td>7185</td>
<td>2020-06-27</td>
<td>2020-06-27</td>
<td>2020-06-27</td>
<td>01:36:30.907</td>
</tr>
<tr>
<td>CLUST</td>
<td>85,424</td>
<td>3829</td>
<td>53,982</td>
<td>1804</td>
<td>2020-07-01</td>
<td>2020-06-30</td>
<td>15:00:15.243</td>
<td>2020-06-27</td>
</tr>
<tr>
<td>HEAP</td>
<td>16</td>
<td>0</td>
<td>2</td>
<td>NULL</td>
<td>2020-06-27</td>
<td>2020-06-27</td>
<td>2020-06-27</td>
<td>02:04:41.130</td>
</tr>
</tbody>
</table>

The data sets for this architecture are formatted as a tuple:

\[(\text{Index\_updates}, \text{access\_date}\,(\text{int}), \text{rows}, \text{index\_seeks} : \text{latency}(\text{output}))\]

The input parameters are: date, rows#, number of hits, equally weighted. The learning technique is supervised by assigning and recording the name, vector data, rows to the latency. Based on the info above, the number of index usage is automatically calculated and also affects the final decision about whether using or not using an index.

The statistics keep additional information for the ANN to be accurate, although our study only focuses on the ones described above in the data set. Our model keeps count of the increase of rows of every table. If either the amount of records is higher than a predefined value or the ANN shows that a merging operation can help the index to reduce the latency, then automatically performs the merging operation between several candidate indexes and then estimate the overall costs of such operations.

The architecture of an ANN deals how the layers are structured, which elements are used in each layer, and how they are designed will often affect the speed and accuracy with which it can perform various tasks. The input layer will take as an observation a list of values [update, access date, rows, the index seeks], being the access date a numeric value that represents the temporal distance from the last time it was accessed and the present time. The number of observations is 35,723. The database for our study is a replica of a development environment with a large amount of activity.

The ANN architecture has a similar format to Reference [21] (Figure 7). It follows a standard topology of three layers, four sigmoid neurons in the input layer. This ANN topology is standard in numerous classification and prediction problems.

**Figure 7.** Neural Network with three layers, four neurons in the input layer.

### 4.2. Training Stage

The weights of the ANN are randomly initialized with normalized values in (0, 1).
During the training phase, we will train the model by initially using 3500 entries in the statistics table. The fitness function is designed to estimate the latency occurring when using the input index. The squared mean error generates the objective function $F$, see Equation (1).

$$F(w_1, \ldots, w_k, \ldots, w_n) = 3500 \sum_{k=1}^{3500} (y_k - \sum_{j=1}^{7} (x_j w_j))^2.$$  \hfill (1)

It is defined as the sum of the squared errors happening when estimating the latency. Due to the high number of observations, the gradient method has been discarded. Gradient descent is the algorithm we have used to train the ANN with a step (learning rate) $\mu = 0.05$. The execution of the algorithms returns the following values for the seven weights, see Table 2.

<table>
<thead>
<tr>
<th>Table 2. Final training configuration of the neural network applied to the dataset. The algorithm output the following values for the seven weight of the neural network.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0166452</td>
</tr>
</tbody>
</table>

The ANN is using 50 epochs, and the training error occurring using the weights above is 0.122667. We have predefined correct predicted value if and only if:

$$\text{abs}(\text{predicted \_ latency} - \text{real \_ latency}) < 0.05 \text{ ms}.$$  

4.3. Testing Stage

The possibility of overfitting our model is low, as the training error is not extremely small. However, cross-validation has been the preferred method to reduce the test error as much as possible. The observations have been divided into 10 data sets. The 10-fold configuration has the following output showed in Table 3.

<table>
<thead>
<tr>
<th>Table 3. Testing results of the neural network applied to the testing dataset.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.137644</td>
</tr>
</tbody>
</table>

As the nature of the problem belongs to a supervised (non-linear) classification, ANN is the preferred model. Alternatively, two more approaches algorithms have been used—K-nearest neighborhood and another ANN, with different topology. Both have been trained and tested. The topology of the new ANN is four input neurons, two hidden layers of 3 sigmoid neurons each, and the output layer (1 neuron). The difference in the previously calculated average errors between the two ANNs is lower than 0.002. The training error difference, minimized by the fitness function is also lower that 0.002, which suggests keeping using the simplest topology for our study. Moreover, The average testing error using cross-validation for K-NN is 0.34665, which is objectively higher than the testing found in both ANN.

4.4. Proposed Algorithm

The algorithms below uses the ANN model for training and testing based on the internal statistics formatted according to the previously defined data sets. Based on the model and the predefined threshold the algorithm decides to use (possibly merge) or not to use a given index, the merging follows the principles of Sections 1 and 3. The algorithm is applied to the collection of tables shown in Section 4.2 and is considering the three types of scenarios. The algorithm detects the situation and uses the index most suitable according to the patterns the ANN found in the data sets.
5. Results

ANNs are, by definition, a non-linear classifier, which suits the nature of the distribution of our data. They have also shown the lowest testing errors when comparing with other algorithms such as K-NN, described in the testing section, that is the main reason why the ANN model is included in the proposed algorithms.

The listing in Algorithm 2 is applied to a simulated environment and compared with a real DBMS. The results have been extracted by randomizing six distributed data sets of hits to a given table. The study compares the use of this algorithm with the merging module versus the traditional algorithm embedded in most DBMS when advising the use of the creation or removal of a given index. The overall latency determines the efficiency of any data that is currently being accessed, which provides a straight forward insight. It compares the latency of accessing random searches in a table with the simulated environment where the same data is obtained by using our hybrid algorithm. Based on the dynamic character of the index generation, the latency of the reads improves. However, the cost of having a learning algorithm embedded in the DBMS exists, and it is part of the system’s overhead. The DBMS implementing this technique checks the structure of their indexes and modifying them if necessary regularly. The simulation has been done by storing the information of the latencies and indexes linked to a set of queries in the standard DBMS combining all possible indexes. Then we have applied the outcome of the ANN model, forcing the DBMS to choose the indexes the ANN recommends for the same queries. We have ignored the index recommendations that are the same as the ones that the standard DBMS uses. Table 4 shows the different ones.

The number of queries used in our study has been increasing with a step $\mu = 10\%$ for every new random distribution (new row). In the initial distribution (row 1), 10,000 queries have been run. The queries are randomly accessing fragment (rows) of the tables of the case of study. The main literals of the queries have been randomly chosen. The simulation automatically generates the ranges of last names and towns. That means that the sets of values of the tables are being randomly elected, offering guarantees of a more robust system.

<p>| Table 4. Comparative costs at different intervals of reads access. |
|-------------------------|-------------------------|</p>
<table>
<thead>
<tr>
<th>DBMS Standard</th>
<th>DBMS Learning Algorithm (Simulation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1121</td>
<td>4.2321</td>
</tr>
<tr>
<td>78.0523</td>
<td>56.9832</td>
</tr>
<tr>
<td>10.1232</td>
<td>17.1031</td>
</tr>
<tr>
<td>11.008</td>
<td>4.7751</td>
</tr>
<tr>
<td>13.1242</td>
<td>13.111</td>
</tr>
<tr>
<td>10.9877</td>
<td>5.2113</td>
</tr>
<tr>
<td>5.4332</td>
<td>6.2323</td>
</tr>
</tbody>
</table>

Table 4 shows the average calculation of the overall costs obtained by the same queries in different batches. The table compares a standard DBMS index optimizer with our intelligent merging based technique. When running this comparative in the target databases, the database buffer pool and cache queries were disabled.

Table 5 compares the sensitivity, also called the true positive rate (TPR) and the specificity (true negative rate—TNR) of the three approaches. It displays the true positives (TP), the true negatives (TN), the false positive (FP), the false negative (FN) and the rates (TPR and TPN) along with the AUCROC, which represents represents TPR vs. False positive rate (FPR). In ML, performance measurement is an essential task. So when it comes to a classification problem, an Area Under The Curve Receiver Operating Characteristics (AUCROC) curve can be used to check or visualize the performance of the multi-class classification problem. It is one of the most important evaluation metrics for checking any classification model’s performance [22].
Table 5. Comparative sensitivity and specificity of the three models.

<table>
<thead>
<tr>
<th>Model</th>
<th>TP</th>
<th>FP</th>
<th>TPR</th>
<th>FN</th>
<th>TN</th>
<th>TNR</th>
<th>AUCROC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANN-1</td>
<td>30256</td>
<td>5467</td>
<td>0.899458</td>
<td>3382</td>
<td>32371</td>
<td>0.855515</td>
<td>0.932716</td>
</tr>
<tr>
<td>ANN-2</td>
<td>30125</td>
<td>5598</td>
<td>0.898690</td>
<td>3396</td>
<td>32357</td>
<td>0.852509</td>
<td>0.923334</td>
</tr>
<tr>
<td>K-NN</td>
<td>27512</td>
<td>8241</td>
<td>0.853770</td>
<td>4712</td>
<td>31041</td>
<td>0.790209</td>
<td>0.867185</td>
</tr>
</tbody>
</table>

Figure 8 depict the comparative results in terms of random scatter searches and by comparative the resulting trends (continuous comparative).

Figure 8. Results in the target database using proposed model. Random search comparative trends.

6. Conclusions and Further Work

As expected, an approach that combines a simple learning technique as regression with a merging operation shows an improvement to the simplistic view that conventional optimizers follow. The advantages are clear as ML techniques in combination with dynamic algorithms are proven extremely efficient in pattern recognition, classification, and prediction, among others. On the other hand, the dynamic merging approach proves to be an efficient technique to use when the cardinality of the index increases. As this problem falls into prediction and classification models, indexing is a great candidate to be used as a target for approaches like this. The results suggest that finding patterns in the index attributes (updates, access date, rows, seeks, and latency) can help assist the optimizer in deciding the most optimal index to use for a given query.

There is a tradeoff for replacing simplistic approaches by more complex ones like our proposal:

1. The redefinition of internal statistics and its need for additional resources, (mainly Virtual Memory and Central Processing Unit)
2. Updates operations are impacted and can also be affected unless the algorithm is designed to work only for SELECT statements.

Further analysis would be needed to see how this affects Data Manipulation Language commands in terms of resource consumption too. This simulation does not take into account the sorting operations and additional memory necessary to perform the merging actions, which in the real production environment can impact the overall queries’ cost. Furthermore, the study has shown figures regarding sole access to the disk. In the future, it could be interesting applying ML techniques to the traditional spatial and temporal locality algorithms [23], that run to place the expected data pages in the buffer pool. Nevertheless, an automated ML and dynamic approach seem to be a promising technique to reduce access time and overall latency when dealing with indexed tables accessed by any data mining algorithms.
Author Contributions: Conceptualization, A.A.A. and N.G.B.; software, L.F.d.M.L. and A.A.A.; investigation, N.G.B., A.A.A. and L.F.d.M.L.; supervision, N.G.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest. This research has no funding sponsors so they had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Abbreviations
The following abbreviations are used in this manuscript:

ANN Artificial neural network
ART Adaptive radix tree
AUC Area under the curve
AUCROC Area under the curve receiver operating characteristics
DBMS Database management system
DMS Database managed spaces
FN False negative
FP False positive
GA Genetic algorithm
K-NN K-nearest neighborhood
ML Machine learning
QO Query optimizer
RDBMS Relational database management system
RID Record identifier
ROC Receiver operating characteristics
SMS System managed spaces
TN True negative
TP True positive
TNR false negative rate
TPR True positive rate

References


