Processing Class-Constraint K-NN Queries with MISP*

Evica Milchevski
TU Kaiserslautern
Kaiserslautern, Germany
milchevski@cs.uni-kl.de

Fabian Neffgen
TU Kaiserslautern
Kaiserslautern, Germany
f_neffgen11@cs.uni-kl.de

Sebastian Michel
TU Kaiserslautern
Kaiserslautern, Germany
smichel@cs.uni-kl.de

ABSTRACT

In this work, we consider processing k-nearest-neighbor (k-NN) queries, with the additional requirement that the result objects are of a specific type. To solve this problem, we propose an approach based on a combination of an inverted index and state-of-the-art similarity search index structure for efficiently pruning the search space early-on. Furthermore, we provide a cost model, and an extensive experimental study, that analyzes the performance of the proposed index structure under different configurations, with the aim of finding the most efficient one for the dataset being searched.

ACM Reference Format:
Evica Milchevski, Fabian Neffgen, and Sebastian Michel. 2018. Processing Class-Constraint K-NN Queries with MISP. In *WebDB'18: 21st International Workshop on the Web and Databases , June 10, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3201463.3201466

1 INTRODUCTION

A classical k-Nearest-Neighbor (k-NN) query returns k objects with the smallest distances to a query object q. In this paper, we address the problem of processing k-NN queries where the result objects should additionally be constrained to specific use-defined classes, C_q. We refer to this problem as a Class-Constraint k-Nearest Neighbor (CCk-NN). This problem is especially common in the domain of geospatial data where we have objects of different classes, like sightseeing, recreational, restaurant, administrative and also more fine-grained types such as cuisine of a restaurant. For instance, consider Alice, who is visiting London. While out for sightseeing, she would like to have lunch in a French restaurant. She is traveling on budget, so the restaurant should be inexpensive. To assist Alice, an application would need to find the nearest k French restaurants, that have good reviews but are also not expensive. The application of *CCk-NN* is not restricted to geospatial data, but it is the most illustrative one.

To solve the CCk-NN problem, we could index the dataset using an index structure suitable for k-NN search, for instance, an R-Tree [5]. To process our k-NN query with class constraints, first a k'-NN query is executed, where $k' > k$. Then, it is checked whether the results fulfill all the classes in C_q. If not enough results are found, the query is executed once again with a larger k', ideally in an incremental fashion. This solution has already been presented by Hjaltason and Samet [6]. The drawback of this approach is that many unnecessary comparisons have to be made, when the classes in C_q have low selectivity, i.e., only a small percentage of objects fall into the classes in C_q. Another possibility is creating an inverted index that maps a class to all objects that fall into this class. Then, instead of going through all objects, we could directly determine those objects that satisfy the classes C_q upfront. Then, we need to compute the distance to the query for all of them, and return the k objects with the smallest distance.

In this paper we present a solution which combines the previous two. An inverted index is used to find the objects that satisfy the classes in C_q, but instead of comparing all these objects to the query, an index structure is used to find the k closest objects to q. Further, we explore and in particular optimize the potential of having inverted indices of different granularity, in the sense that not only single classes but pairs, triplets, etc. of classes are used as keys to determine the corresponding objects.

1.1 Problem Statement

Given a set of objects $O = \{o_1 \dots o_n\}$, where each object $o_i \in O$ has a set C_{o_i} of associated classes from a global set of classes $C = \{c_1, \dots, c_j\}$, and a distance function d, the task is, for a user specified query q and query classes $C_q \subseteq C$ to find k objects $O_R \subseteq O$ such that $\forall o_r \in O_R : C_{o_r} \supseteq C_q$ and there is *no* object $o_s \in O$ and $o_r \in O_R$ such that $d(q, o_s) < d(q, o_r)$ and $C_{o_s} \supseteq C_q$.

If there is more than one class constraint given ($|C_q| > 1$), then they are all applied, thus they have a conjunctive meaning. The result set can be smaller than k in the case when we have less than k objects satisfy the classes in C_q. Each class is of the form attribute:value, for instance color:blue.

As with standard k-NN search, we aim at having as few as possible object comparisons, low memory usage, fast index creation time, while having low query response time.

1.2 Contributions and Outline

The contributions of this paper are as follows: *(i)* We propose an index, coined MISP, to efficiently solve the CCk-NN. In MISP, objects that belong to the same class, or combination of classes, are indexed together using a separate similarity-search index like an R-tree, which we call sub indices[1]. In addition, MISP integrates an inverted index to easily match the classes, or combination of classes, to the sub indices. One major challenge that arises from our indexing approach is the question for how many and for which classes, or combination of classes, do we create sub indices. Since one object can belong to several classes, it would also be indexed by multiple indices. The number of sub indices that can be created, and thus the memory and construction time overhead, quickly increase as the set of classes increases. Thus, *(ii)* we present a cost model which estimates the performance and index size of the MISP index.

*This work has been supported by the German Research Foundation (DFG) under grant MI 1794/1-1 and MI 1794/1-2

WebDB'18, June 10, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-5648-0/18/06…$15.00
https://doi.org/10.1145/3201463.3201466

[1] The general problem setup and potential of naively materializing all sub indices was already demonstrated in own prior work [3].

Furthermore, we sketch an algorithm and discuss how a configuration of MISP can be chosen that would perform best under limited memory resources. *(iii)* Using both a synthetic and a real dataset, we first experimentally evaluate the correctness of the proposed cost model, and, second, we evaluate the performance gain of the proposed index against existing approaches under different configuration setups.

The paper is organized as follows: The related work is presented in Section 2. Section 3 presents our combined index, how it is constructed and queried. In Section 4, we provide a cost model that estimates the performance of the index depending on the number of sub indices created. In Section 5, we show a detailed experimental evaluation with both synthetic and real dataset, where we also show the accuracy of our cost model. Finally, in Section 6, we conclude the paper.

2 RELATED WORK

Hjaltason and Samet [6] propose an algorithm that can compute the $k + n$ nearest neighbors, if needed, without accessing the already searched data again, i.e., the processing is incremental. In order to do so, they employ a modification of an R-Tree [5]. Another indexing strategy that allows incremental searching is iDistance [7]. It utilizes the idea that searching for the k nearest neighbors can be done by only using their distance to some reference point O_i. Thus, they map the objects into linear space, by indexing, and afterwards searching, their distances to this reference point O_i. In order to find the k nearest neighbors to a query q, an incremental range search is done over the B+-Tree. The index works for metric space, because the idea of searching by distance to a reference point uses the triangle inequality. The proposed solutions can be applied to our problem, but they have the drawback that all the data needs to be searched for a query q, even though large percentage of the data could have no chance of qualifying as a result. Combining the benefits of an inverted index and an R-tree has already been applied for solving the problem of location-based web search [1, 2, 4, 9]. The main task of this problem is to determine documents that are in terms of content and location relevant to the query. To solve this problem they combine information retrieval techniques with nearest neighbors search. Milchevski et al. [8] combine an inverted index with a metric index structure to solve the problem of finding similar top-k rankings. Their index, called Coarse Index, combines an inverted index with a BK-tree. Although our approach also combines inverted indices with a tree-based indexing structure, we specifically address the problem of handling sub indices induced by classes that are associated with the objects.

3 MULTI-KEY INVERTED INDEX WITH SMART POSTING LISTS (MISP)

In this paper, we combine an inverted index with existing k-NN search indexing technique, into a new index, named Multi-Key Inverted Index with Smart Posting Lists (MISP). MISP combines the benefits of both inverted indices and k-NN index structures. This way, we can easily find the data objects that belong to the classes we are searching for, and then find the most similar k ones without comparing them all to q. Figure 1 illustrates this index. Instead of having data stored in plain posting lists, we index them using iDistance [7], an index structure for searching data in metric space that allows incremental k-NN searching—but any other index structure for k-NN search can be used instead. The MISP index can also have

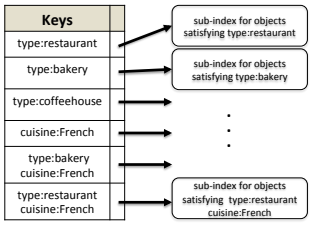


Figure 1: Multi-Key Inverted Index with Smart Posting Lists

combinations of classes as keys in the inverted index. Thus, only the objects that satisfy all of a key's classes are indexed in the corresponding iDistance sub index. We call these m-key sub indices, where m is the number of keys for this sub index. Figure 1 illustrates simplified MISP index for the Alice example presented in Section 1. This MISP index has indexed in 1-key sub indices all objects that are of type restaurant, coffeehouse, or bakery, as well as objects with cuisine French. In addition, via 2-key sub indices, both French restaurants and French bakeries are indexed. Hence, when Alice is searching for the nearest French restaurant, we can directly search for the nearest French restaurant using the created 2-key sub index—instead of searching all the restaurants and then pruning the ones which are not French.

However, there is also the drawback that a French restaurant will be indexed three times: once for restaurants, once for objects with French cuisine, and once in the sub index for French restaurants. However, when searching for French coffeehouses, we have the opposite situation: During search, we have to access either the sub index containing all coffeehouses, or the one for French cuisine, but we do not have an additional storage overhead. If we do not expect many queries for French coffeehouses, this would be the preferred solution.

Index Creation: MISP is built layer-by-layer in a *bottom-up* manner. First, we create the 1-key sub indices. For correctness reasons, in order not to miss any potential result object, this layer is completely built, i.e., there is one sub index for each class $c \in C$. This layer represents a foundation for the next layers and also serves as *fallback* for every query, because every data object must fall into one of these indices. For instance, if we are given a query q with two class constraints, restaurants and French, $|C_q| = 2$, it can happen that there is no sub index created for this combination of classes. Then, to find the results, we can query the single-key sub indices, either the one for restaurants or for French objects depending on its size. For the next layer(s), the 2-key sub indices (and 3-key sub indices, etc.), we can decide, if we want to build all possible ones. We combine every 1-key sub index with every other. In order to speed up construction, we use the layer below as basis for calculating the possible combinations. The drawback of this approach is that we have to have all possible combinations from the layer below, to make every possible combination in the current level. If we do not build the level of $m - 1$ fully, we will miss some combinations in the m-key sub indices. The maximum m for which we create m-key sub indices we refer to as the *depth of the MISP index*. The depth of the MISP index in Figure 1 is two.

Querying the MISP index is slightly different than querying the inverted index. Instead of querying all sub indices for each class in C_q, and then merging the results, we query only the sub index for the most selective class (combination of classes) in C_q, and the sub

index is incrementally searched until we find k nearest objects that satisfy all the constraining classes. This is possible since each object in the index has a set of classes that it belongs to, C_{o_i}, attached to it. In order to find the most selective class, during construction time of the index, we gather statistics for how many elements each sub index has indexed and, we keep class co-occurrence statistics. When at query time we are given a set of classes C_q where we have a combination of classes that do not occur together in the dataset, we terminate the search early, as there are no possible results.

When we have MISP index with larger depth, finding the most selective sub index imposes a small overhead, because in order to find it, we first create the power set of the query's classes, that is, 2^{C_q}. Then, for each element of this power set, we check if we have the corresponding sub index and, if so, its size.

Therefore, in order to reduce the overhead, while increasing our chances to "hit" a selective multi-key sub index, we create only those multi-key indices of the classes that contain the most elements in the dataset D. The idea behind this is that we assume that the issued queries would follow the same class distribution as our dataset D. In order to find the most popular classes, we use the class co-occurrence statistics mentioned above. If the queries are not following the dataset characteristics, one would require a query log for analysis, but the idea would remain the same.

Since an object can take on multiple classes, the drawback of MISP is that its memory consumption as well as construction and maintenance overhead increases with the number of sub indices created. Furthermore, the time needed for searching for the best most selective sub index increases, as we have to check for every m-combination of class constraints, if they exist and if yes, how many objects are in this sub index. Even when creating the multi-key sub indices only of the most popular classes, the number of possible combinations of sub indices, computed as $\sum_1^c \binom{c}{i}$, drastically increases with the number of possible classes (i.e., $|C| = c$). To address this issue, in the next section, we present a cost model that finds the cutting point at which the creation of additional sub indices would lead only to negligible performance gains.

4 COST MODEL

4.1 Cost for Querying

In order to understand the cost for querying MISP, we compute the expected number of distance function calls, for a given query q and a set of constraining classes C_q. For this, we need to estimate the size of the sub index, the probability of hitting an index structure at some index level m, and the cost for querying the specific sub index structure used. However, since our goal is to devise a general and simple cost model, independent of the underlying index structure used, we devise a cost model where we assume that all indexed objects in the queried sub index are evaluated via a full scan. This gives us an estimation of the upper bound of the cost for querying the MISP index. Furthermore, for simplicity, we assume that the data is uniformly distributed over the global set of classes C, and we assume that the size of the dataset $|D| = d$ and of the global set of classes $|C| = c$ is known. We also assume that the average expected size of the set of constraining classes $|C_q| = a_q$ is known. We start by estimating the cost for querying a MISP index with depth 1, which is basically an inverted index where each sub index contains elements from only one class.

Let Y be a random variable representing the number of objects, y_i, in the sub index for the i-th class c_i. $E[Y]$ is then the expected number of data points in this index. When the objects in the dataset belong to more than one class, they will be inserted into every sub index they belong to. Therefore, to estimate the expected size of the sub indices, we need to estimate the average count of classes that an object belongs to, avg_{c_o}. This can be computed as $avg_{c_o} = \sum_i \frac{C_{o_i}}{|O|}$. The total number of stored objects in the MISP index can be estimated as $d \cdot avg_{c_o}$. Then $y_i = \frac{d \cdot avg_{c_o}}{c}$ and $P(y_i) = \frac{1}{c}$ as we are always querying only one sub index in MISP. Thus, the expected size of the queried sub index can be computed as $E[Y_{InvSeq}] = \sum_{i=1}^{c} y_i \cdot P(y_i) = \sum_{i=1}^{c} \frac{d \cdot avg_{c_o}}{c} \cdot \frac{1}{c} = \frac{d \cdot avg_{c_o}}{c}$. Thus, in the case of having a sub index only for each individual class, $E[Y_{InvSeq}] = \frac{d \cdot avg_{c_o}}{c}$ would be the upper bound on the number of distance function computations.

Next, we extend the cost model to estimate the cost of a MISP index of depth 2, where in addition to the sub indices for each class, the MISP index also contains sub index for each pair of classes in C. In this case, we need to take into account the number of additional sub indices created. This can be calculated as $max_{extra} = \binom{c}{2} = \frac{1}{2}c(c-1)$. Since in practice the number of max_{extra} sub indices can be very large, we actually would create only $l, l \leq max_{extra}$ sub indices. To calculate the expected number of items in such an index, we first need to estimate the expected size of a 2-key sub index. Similarly to the expected size of the single-key sub indices, we have $P(y_i) = \frac{1}{l}$ and $y_i = \underbrace{\frac{d \cdot avg_{c_o}}{c}}_{\text{first "entry"}} \cdot \underbrace{\frac{avg_{c_o} - 1}{c-1}}_{\substack{\text{matching} \\ \text{second "entry"}}}$.

Then:

$$E[Y_{2-key}] = \sum_{i=1}^{l} y_i \cdot P(y_i) = \sum_{i=1}^{l} \frac{d \cdot avg_{c_o}}{c} \cdot \frac{avg_{c_o} - 1}{c-1} \cdot \frac{1}{l} \quad (1)$$

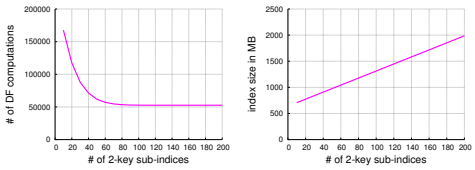
Next, we need to estimate the probability, that we hit such 2-*key sub index*. The probability depends on the number of class constraints $|C_q|$ provided at query time. If we have only one class constraint on the query, we can not hit a 2-*key sub index* at all. If we have built all possible 2-*key sub indices* and $|C_q| \geq 2$, we have a 100% chance that we will hit such sub index. This is estimated with the following formula:

$$P(2-key) = \begin{cases} 0 & a_q = 1 \\ 1 & l = max_{extra} \wedge a_q \geq 2 \\ \frac{l}{max_{extra}} & a_q = 2 \\ 1 - B_{n,p,k}(\binom{a_q}{2}, \frac{l}{max_{extra}}, 0) & a_q > 2 \end{cases} \quad (2)$$

$B_{n,p,k} = \binom{n}{k} p^k (1-p)^{n-k}$ is the probability to hit exactly k times in a Bernoulli process with n trials and p is the success probability. In our case, we have a Bernoulli process where n is the number of pair of classes that we can build with the classes in C_q. Then, we want to know, which are the chances that these would match at least one of the 2-key sub indices that are created. This is expressed by the complementary probability of hitting exact zero times.

The final expectation of data point in the accessed index is:

$$\begin{aligned} E[Y_{MISP_{2-key}}] = {} & P(2-key) \cdot E[Y_{2-key}] \\ & + (1 - P(2-key)) \cdot E[Y_{InvSeq}] \end{aligned} \quad (3)$$



(a) (b)

Figure 2: Cost estimation (#df computations (a) and index size in MB (b)) for 2-key MISP index when varying the number of 2 key sub indices created ($d = 1,000,000$, $c = 20$, $avg_{c_o} = 5$ and $a_q = 5$).

Estimating the cost at any level m can be done accordingly and is omitted due to lack of space.

By plugging a value for the number of created sub indices l at level m, l_m, in Equation 3 we can estimate the number of distance function computations depending on the number of sub indices created at each level of the MISP index. Thus, Equation 3 should ideally help us not only in choosing the depth of the index, but also the number of sub indices created at level m.

According to Equation 3, Figure 2a shows the estimation of the number of distance function computations when varying the number of 2-key sub indices for a dataset with the following parameters: $d = 1,000,000$, $c = 20$, $avg_{c_o} = 5$ and $a_q = 5$. As expected, the number of distance function computations is decreasing as we increase the number of sub indices created. However, we can see that the rate at which the value is decreasing is dropping, and after some point, in this case around $l = 60$, the decrease in distance function computations becomes insignificant. One way of choosing the number of sub indices to be created at level m would be to calculate the percent decrease and then stop when this becomes lower than some threshold value t. Below, we propose an algorithm for choosing the number of sub indices to be created that also takes into account the estimated size of the index.

4.2 Estimating the Size of the Index

The size of the MISP index can be easily estimated using the equations from before. Note that in this approximation, the occupied space for the data itself will be left out, because it will always be constant for one data set, independent of the employed indexing method. This means, when comparing two different index structures, only the difference of their overhead is relevant. We try to formulate this as general as possible, as this overhead depends heavily on the implementation.

For estimating the size of the MISP index we again need the expected size of the sub indices, the size of |C|, and the overhead imposed by the index structure. Let X be a random variable representing the size of the sub indices. Without loss of generality, we assume that in the index we only store pointers to the data objects. For the MISP index with depth one we would have: $E[X_{MISP_1}] = c \cdot (oh_{tree} + pointer + string + integer) + c \cdot \frac{d \cdot a_d}{c} \cdot oh_{node}$, where the string and the pointers take into account for storing the key in the inverted index, and a pointer to each sub index, and oh_{tree} and oh_{node} account for the overhead of each sub index and each node in the sub index which we consider them to be trees.

Generalizing this cost model to level m is straightforward, keeping in mind that we can reuse the estimations from the previous subsection. We have:

input: $|D|$, $|C|$, a_q, avg_{c_o}, $step$
1 **while** $m \leq a_q$ **do**
2 $l_i = 0$; increase(m,1)
3 **while** improvement(L,M) > cost(L,M) **do**
4 increase(l_i,$step$)
5 M=est_size(m, l_i);
6 L=est_df_computations(m, l_i)
7 **return** L, m

Algorithm 1: Algorithm for determining the depth m and the number of sub indices per level for the MISP index by accounting the performance gained versus the memory consumed for the created sub indices.

$$E[X_{MISP_m}] = (l_m \cdot (oh_{tree} + pointer + i \cdot string + integer) + l_m \cdot ((d \cdot \prod_{i=0}^{m-1} \frac{a_d - i}{c - i} + 1) \cdot oh_{node}) \quad (4)$$

where $1 \leq l_m \leq \binom{c}{m}$ is the number of sub indices created at level m. To get the total size of the MISP index we just need to sum up the costs from each level. Figure 2b shows the estimated cost for a 2-key MISP index for a dataset with $d = 1,000,000$, $c = 20$, $avg_{c_o} = 5$ and $a_q = 5$. Furthermore, as overhead for the tree and node we have used $oh_{tree} = 704b$, $oh_{node} = 1024b$, $pointer = 256b$, $string = 576b$ and $integer = 128b$, which account to our Java implementation. As expected, the size of the index increases linearly with the number of sub indices added (cf., Figure 2b).

4.3 Overall Cost

To put the two costs together one needs to consider the trade-off between the performance gain induced by having more sub indices and the price of increased memory consumption. At some point, as it can be seen in Figure 2, the reduction in distance function computations becomes marginal, while the memory consumption steadily increases. At this point, the price paid, in terms of memory consumption, for creating more sub indices is larger than the benefit gained.

Algorithm 1 sketches this idea. For each level m, the distance function computations and the index size is estimated, for varying number of sub indices created. While the benefit from having these indices is higher than the cost paid, we continue adding more indices. Once this stopping condition applies, the number of sub indices that should be created for the current level is marked, and we continue with the estimations for the next level m. How exactly the trade-off between the benefit gained and the cost paid can be computed is left for future work.

Depending on the use case scenario, as cost paid can be considered also the construction time, or the overhead imposed for updates. Considering that when adding more sub indices we also increase the redundancy of objects in the MISP index, the cost for updates needs to be considered. While the construction time can be deduced form the size of the index, the cost for updates needs more detailed investigation. This, however, is beyond the scope of this paper.

5 EXPERIMENTS

All experiments were executed on a desktop machine running Ubuntu 16.4 with an Intel Core™ i5-4570 CPU and 8 GB of RAM (5 GB used

by the Java VM). We implemented the different indexing strategies using Java 8. We have focused on measuring the performance of the following indexing strategies:

- Sequential scan (Seq), as a baseline approach
- *iDistance* [7] having all the data objects indexed.
- Inverted index using single classes as keys (*InvSeq*).
- Our combined index, MISP, where the elements in the inverted index are evaluated by sequential scan (MIPS-seq).
- Our combined index, MISP, where iDistance is used as sub index (MISP).

To get some insights into the performance of the different indexing strategies we measured the following: *(i)* index creation time, *(ii)* memory consumption, *(iii)* wallclock time, and *(iv)* number of distance function (df) computations. We report the time (df computations) needed to execute 100 queries, with $k = 20$.

Datasets: For the experiments we used both a synthetic and a real dataset. The synthetic dataset consist of two dimensional points where the value of every dimension is fixed between 0 and 100. The dataset is created by first randomly picking points that act as the pivot of the partitions. Then points in each partition are added by moving the points away from the pivot. The direction, and the distance is chosen randomly, however, it does not exceed the radius of the partition, provided at input. Important to note is that the data objects are distributed uniformly over the classes. For the experiments in this paper, we have generated a dataset with the parameters: $|O| = 1,000,000$, $|C| = 20$, $|C_{o_i}| \leq 5$, $|C_q| \leq 5$.

For the real-world dataset, we used geospatial data provided by OpenStreetMap (OSM). We used the available data from within the boundaries of Germany[2]. For the experiments we focus on Points of Interest (POIs), and the rest of the data, e.g., streets, railroads or buildings, were dropped. We were left with about 1.3 million data entries. The data in the OSM dataset is described by a set of user specified tags whose values are freely inputed by users. The tags we cleaned and adapted as classes of the data objcets. To each data object we assigned at most six classes, thus $\forall o_i \in O : |C_{o_i}| \leq 6$: amenity, shop, city, hasName, hasOperator, hasOpeningHours. The first two classes can take many values. *Amenity* can take more than 8500 and *shop* more than 10000 different values. For city the values are only sometimes there. The last three we made them as boolean classes, an object either belongs to this class or not. For instance, a data object with the tags "amenity=pub, name=Wladi Rockstock, opening_hours=Tu-Su 18:00+" would get the classes "amenity:pub", "hasName" and "hasOpeningHours". The queries that we use for the experiments are randomly chosen from the data.

Validity of the cost model: In order to assess the validity of our cost model, we conduct an experiment where we measure the performance of the MISP index as we vary the number of 2-key and 3-key sub indices created. For this experiment, we used the synthetic dataset described above. We also measured the performance for a version of the MISP index, where the sub indices are plain posting lists, denoted as MISP-seq. We did this because this better reflects the cost model, which estimates the distance function comparisons to all the objects in a sub index.

As we can see in Figure 3, the cost model provides an accurate estimation of the real performance and size of the MISP index for level 2. The performance of the MISP index where as sub index we use the iDistance index is lower than the one estimated

[2]http://download.geofabrik.de/europe/germany.html, downloaded on 2016-11-20

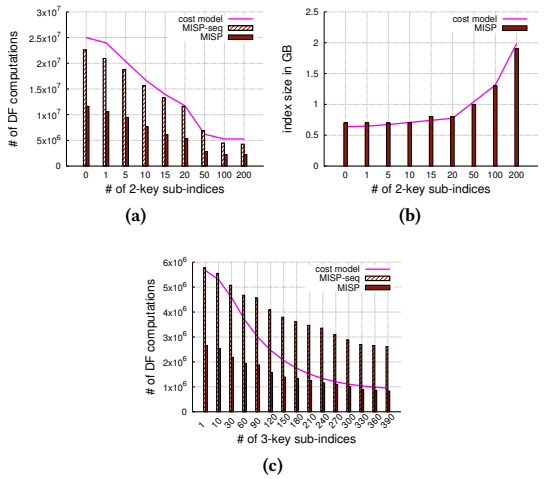


Figure 3: Comparison of the cost model estimation with the experimentally measured df computations (a) and index size (b) when changing the number of 2-key and 3-key (c) sub indices (synthetic dataset).

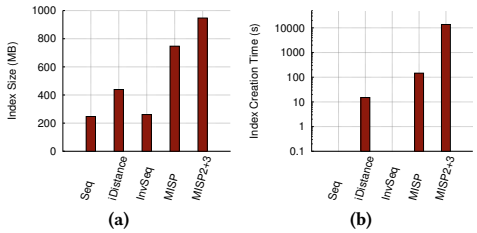


Figure 4: Memory consumption (a) and creation time (log-scale) (b) of different indexing strategies (OSM dataset).

with the cost model. However, what is important is that the performance improvement when adding more sub indices follows the same trend. For the estimation for the number of sub indices at level 3 (Figure 3c), we have created only 60 sub indices at level 2, as this seems to be the point at which the improvement in distance function computations becomes marginal. We see that the cost model at first overestimates the decrease in the number of distance function computations, but then at around 200 sub indices, the cost model starts to more realistically capture the decline in distance function computations of the MISP index.

Comparison of different indexing techniques Next, using the OSM dataset, we compare the MISP index against the inverted index and the iDistance. We also plot the result when having a plain sequential scan, as a baseline. To see whether introducing 3-key sub indices would lead to more performance benefits, we have also created a version of the MISP index with the best 500 2-key iDistance sub indices, and the objects belonging to the remaining 2-key combinations indexed in a corresponding posting list, instead of using an iDistance index. In addition to this we also created 500 3-key iDistance sub indices. This MISP index version is denoted as MISP2+3 in Figures 4 and 5 .

Figure 4 shows the memory consumption and the creation time of the MISP where we have created the best 500 2-key sub indices, compared to the iDistance the inverted index and a sequential scan.

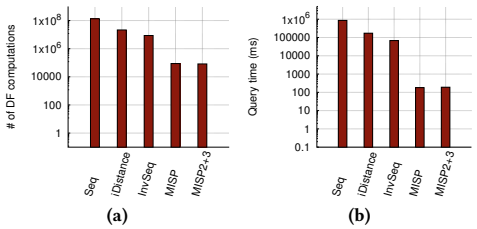


Figure 5: Log-scale plots of df computations (a) and query execution time (b) different indexing strategies (OSM dataset).

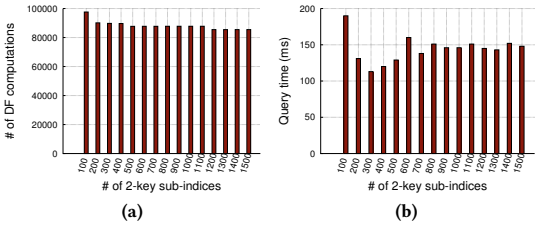


Figure 6: df computations (a) and query time (b) of MISP index for different number of 2-key sub indices (OSM dataset).

The reason for creating 500 sub indices is explained below. As expected, the creation time and memory consumption of MISP index is worse compared to the two baselines. Especially for the creation time, we see that creating additional sub indicies presents quite some overhead, especially when we have created all 2-key sub indices and 500 3-key sub indices (MISP2+3). However, even though we are working with relatively large dataset of 1.3 million objects, the memory consumption is still reasonable, i.e., under 1GB.

Figure 5 shows the distance function computations and the query execution time for different indexing techniques. As we can see in Figure 5a by using the MISP index with 500 2-key sub indices we have significantly lower number of distance function computations compared to the iDistance index or the inverted index. Accordingly, the MISP index has the best query execution time as well (Figure 5b). It is interesting to notice that the inverted index outperforms the iDistance index for this dataset. We can see that adding the additional 3-key sub indices only slightly reduces the number of distance function computations. There is even a slight degradation in the query execution time due to overhead for finding the best index. Note, that Figure 5 is in logarithmic scale.

Varying number of 2-key sub indices using the real dataset: Next we evaluate the performance of the MISP index using the OSM dataset when varying the number of 2-key sub indices. Only for 2-key sub indices in this dataset there are 196, 284 possible combinations. Note that all the 1-key sub indices are also present in the MISP index, and we create the n best 2-key sub indices as described in Section 3. Figure 6a shows that for the real dataset the distance function computation follow a similar trend as with the synthetic dataset. At first there is a drastic drop in the number of distance function computations, which then starts to slowly decrease. However, in Figure 6b we see that as we increase the number of 2-key sub indices there is an added overhead from finding the best index, and thus the query execution time at first improves, but then starts to slightly deteriorate. At the moment this overhead is not taken into account in our cost model. This can be done in future work.

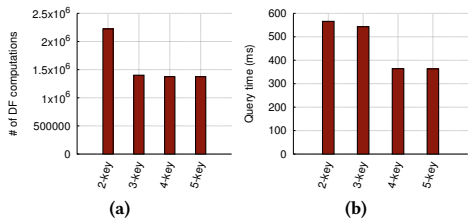


Figure 7: Comparison of df computations (a) and query execution time (b) when different levels of m-key sub indices are build (synthetic dataset).

We see that for 500 2-key sub indices the df computations only marginally improve, and there is no significant performance degradation. That is why in the previous experiment only the best 500 sub indices were created.

Performance based on the depth of MISP index: To better investigate the performance of the MISP index, when having an index with different depth, i.e., when increasing the value of m we performed an experiment using the synthetic dataset. We test the performance of the MISP index for values $2 \leq m \leq 5$. However, to reduce the overhead, on each level we create only the 100 best m-key sub indices. The performance of the MISP index, as it can be seen in Figure 7, does not improve much as we add additional layers. We have measured that the construction time increases significantly with adding layers of m-key sub indices. The memory consumption, on the other hand, shows not to be such an issue.

6 CONCLUSION AND OUTLOOK

In this work, we proposed and analyzed an indexing strategy for solving the Class-Constrained k-NN problem. As a solution we proposed an index that combines an inverted index with a similarity search index. To solve the problem imposed by the proposed indexing strategy, of how many and which sub indices do we need to create to get the best performance, while reducing the overhead, we proposed an experimentally validated cost model Furthermore, in an experimental study we show that our indexing strategy performs better over existing indexing techniques. As a future work, we would like to extend our data model to include class types, and class instances, where not all possible class combinations would be possible. Taking into account the query history when choosing the sub indices to be created could also be investigated.

REFERENCES

[1] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel. Text vs. space: efficient geo-search query processing. *CIKM'11* pages 423–432, 2011
[2] G. Cong, C. S. Jensen, and D. Wu. Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects. *PVLDB* volume 2, pages 337–348, 2009,.
[3] J. A. de Souza, A. J. M. Traina, S. Michel. Class-Constraint Similarity Queries. *SAC*, 2018
[4] I. De Felipe, V. Hristidis, and Naphtali Rishe. Keyword Search on Spatial Databases. *ICDE*, pages 656–665, 2008. 656–665.
[5] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84*, pages 47–57, 1984.
[6] G. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *ACM Trans. Database Syst.*, pages 265–318, 1999.
[7] H. V. Jagadish, B. C. Ooi, K. Tan, C. Yu, and R. Zhang. iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, pages 364–397, 2005.
[8] E. Milchevski, A. Anand, and S. Michel. The Sweet Spot between Inverted Indices and Metric-Space Indexing for Top-K-List Similarity Search. In *EDBT*, pages 253–264, 2015.
[9] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W. Ma. Hybrid index structures for location-based web search. *CIKM*, pages 155–162, 2005.