Nearest Neighbor Queries with Counting Aggregate-based Conditions

Daniel S. Kaster[1,2,*], Willian D. Oliveira[2], Renato Bueno[3], Agma J. M. Traina[2] and Caetano Traina Jr.[2]

[1] Department of Computer Science, University of Londrina (UEL), Londrina, PR, Brazil
dskaster@uel.br
[2] Department of Computer Science, University of São Paulo at São Carlos (ICMC-USP), SP, Brazil
{willian, agma, caetano}@icmc.usp.br
[3] Department of Computer Science, Federal University of São Carlos (UFSCAR), SP, Brazil
renato@dc.ufscar.br

Abstract. The amount of complex data, such as images, videos and time series, has been increasing in the past few years in a very fast pace. The main property employed to query such kind of data based on its content is the similarity between elements. One of the most common similarity queries is the k-Nearest Neighbor query (k-NNq), which returns the k elements most similar to a given reference element. Although this kind of query is useful for many applications, it does not consider additional conditions that may modify the basic nearest neighbor search, which would allow retrieving more relevant information to the user. Complex data are usually associated with other information and usually have metadata describing the data content and/or context. Such external (non-content-based) information can be employed to define the conditions modifying the k-NN search to answer advanced queries over complex data. This article presents a new variation to the k-NNq, which includes counting aggregate-based conditions. This extension allows answering queries that are not easily definable using only external conditions and regular k-NN operations. Our work defines the counting aggregate-based conditions and presents algorithms to execute k-NN queries with these conditions, using either sequential scan or a metric access method. Experiments over real datasets are also presented, comparing the developed algorithms and showing that search performance can be largely improved using an appropriate access method.

Categories and Subject Descriptors: H. Information Systems [**H.m. Miscellaneous**]: Databases; H. Information Systems [**H.3. Information Storage and Retrieval**]: H.3.1. Content-Analysis and Indexing

Keywords: Metric Access Methods, Nearest Neighbor Search, Similarity Queries

1. INTRODUCTION

With the increasing dissemination of devices to capture digital data, such as digital cameras, medical scanners, satellites and biomolecular machines, huge amounts of complex data are being generated every day. In this article, *complex data* are those data that cannot be represented using simple/traditional data types, such as numbers, dates and short strings. To allow querying such data, a set of features can automatically be extracted from the data. Retrieving complex data based on these features is known as *content-based retrieval*. The set of features extracted is called a *feature vector*, or *signature*, which is used in place of the original data to evaluate queries. Comparisons over complex data are based on (dis)similarity relations between pairs of feature vectors, therefore, the queries that employ such comparisons are known as *similarity queries*. The most common similarity queries are the Range query (Rq), which returns the stored elements whose dissimilarity to a given query reference is less than or equal to a threshold, and the k-Nearest Neighbor query (k-NNq), which returns the k elements that are the closest to the query element [Barioni et al. 2011].

*Corresponding author. On leave at Dept. of Computer Science, University of São Paulo at São Carlos, SP, Brazil.
This work has been supported by CNPq, CAPES, FAPESP and Microsoft Research.

Data structures and algorithms to answer similarity queries have been extensively addressed in the literature [Hjaltason and Samet 2003]. However, most of the existing works consider similarity operations isolated from other filter conditions. Complex data frequently have associated metadata, which can be employed to define queries. For example, geographical applications usually associate information to a stored map. Therefore, a public area can be represented as a polygon with touristic information associated to it, and a hotel can be a point whose metadata include its amenities and room rates. Other examples are the images of medical imaging exams. They are part of the patient record, which stores information about the patient's history of health and medical care. Furthermore, medical images themselves store metadata about the patient, the exam and the imaging equipment, as they usually follow the DICOM (Digital Imaging and Communications in Medicine) standard, which defines a rich set of attributes in the image header to describe the image context.

A k-NN query is represented through a variation of the relational selection, represented as $\ddot{\sigma}_{selection\_predicate}(R)$, where the selection predicate has the form $\langle S\ k\text{NN}[\delta, k]\ s_q\rangle$, being S an attribute, in the domain $\mathbb{S}$, of the relation R, $k\text{NN}[\delta, k]$ the k-NN operator and $s_q \in \mathbb{S}$ the reference element [Barioni et al. 2011]. Therefore, the query $\ddot{\sigma}_{S\ k\text{NN}[\delta,k]\ s_q}(R)$ returns the tuples whose attribute S stores an element that is one of the k nearest neighbors of s_q, regarding the distance function δ. This definition allows combining a $\ddot{\sigma}$ with the remainder relational operators in a query. However, there still are neighbor-based queries that cannot be represented using a regular k-NN operator in the $\ddot{\sigma}$ predicate. For instance, suppose that one wants to issue Query following, over the table $Cities(name, pop, coord)$, which stores the names, the populations and the coordinates of cities.

Query 1: *Return the* 20 *cities that are the closest to Bristol city, USA, such that the result includes at least* 5 *cities having a population of* 100,000 *or more people.*

The nearest neighbor search that answers this query is constrained by a condition that cannot be stated as a selection neither before nor after the k-NNq. That is, Query 1 is not answered by $\sigma_{pop\geq 100{,}000}(\ddot{\sigma}_{coord\ k\text{NN}[L_2,20]\ BristolCoord}(Cities))$, where L_2 is the Euclidean distance and $BristolCoord$ is the coordinates of the Bristol city, because this statement only returns the cities having 100,000 or more people that are among the 20 Bristol's nearest neighbors. The query is not answered by $\ddot{\sigma}_{coord\ k\text{NN}[L_2,20]\ BristolCoord}(\sigma_{pop\geq 100{,}000}(Cities))$ either, as this statement only returns cities with population greater than or equal to 100,000. In fact, Query 1 may require discarding cities having less than 100,000 people that are closer to Bristol city to include at least 5 cities with a population of 100,000 or more people. This kind of condition is best computed during the k-NN search, as it modifies the behavior of the basic search. In this article, we call this kind of condition as a *aggregate-based similarity condition*, as it depends on an aggregation function computed over the elements that compose the similarity-based query answer. More specifically, Query 1 relies on a condition involving a filtered count of the tuples in the result, which we refer herein as *counting aggregate-based similarity conditions*, or *ca-cond*.

Another example of query with a similar reasoning is Query 2, defined over a georeferenced database of hotels as follows.

Query 2: *Return the* 10 *hotels that are the closest to the conference meeting place, such that at most* 2 *of them are* 10 *miles or more far from the airport.*

This query may also return hotels that are farther than the k-th closest hotel to the conference meeting place, in order to satisfy the given condition. Furthermore, notice that this is an example of a k-NN query constrained by a *ca-cond* employing another similarity-based criterion (the distance from the airport).

An additional example involves medical images databases. Medical exams, such as computed tomography (CT) and magnetic resonance imaging (MRI), usually organize the series of images that compose an exam into what the radiologists call a study. In most situations, the physicians are inter-

ested in the study as a whole, seldom analyzing an image out of the study context. Therefore, when issuing a k-NNq over the exam image database, they usually want to define the number of elements of the result, i.e. the value of k, as a function of the number of studies and not as the number of individual images, as stated in Query 3, stated as follows.

Query 3: *Return the* 10 *medical images that are the most similar to the query image i_q, such that every image returned is from a distinct exam study.*

The result of Query 3 should include only the image that is the most similar to the query image in each exam study, maybe discarding images that are more similar to i_q than other images in the result, in order to have images from 10 distinct studies.

The objective of this article is to introduce a new variation to the k-NN query, called the condition-extended k-NN query ($_ck$-NNq). The $_ck$-NNq allows indicating an assertion modifying the nearest neighbor search, which can be a tuple-based condition or an aggregate-based similarity condition. In this work we focus on a particular kind of aggregate-based similarity condition: the counting aggregate-based condition. We define counting aggregate-based similarity conditions, present the main algorithms to solve them and show experiments evaluating the proposed algorithms.

This article is structured as follows. Section 2 summarizes the concepts needed to understand our approach and existing related works. Section 3 introduces the new $_ck$-NNq operation. Section 4 defines the counting aggregate-based conditions to the $_ck$-NNq, while Section 5 shows the proposed algorithms to execute counting aggregate-based $_ck$-NN queries. Section 6 presents experiments comparing the developed algorithms and Section 7 presents the conclusions and future work.

2. BACKGROUND AND RELATED WORK

The nature of similarity queries precludes using conventional indexes based on the total order property to speed up their execution. *Metric Access Methods* (MAMs) are the most adequate data structures to index complex data. MAMs represent data as elements in a *Metric Space*, which only require the elements and their pairwise distance [Zezula et al. 2006]. Therefore, a MAM is able to efficiently index data whenever a distance function is defined, such as (high-)dimensional feature vectors as well as feature vectors that do not have a defined dimensionality, such as the features generated by extractors recognizing regions of interest in an image, whose number can vary for each image. Examples of efficient dynamic MAMs are the M-tree [Ciaccia et al. 1997; Skopal and Lokoč 2009] and the Slim-tree [Traina Jr. et al. 2002], which improved the M-tree adding functionalities to measure and to reduce the overlap between subtrees.

The most known methods to execute k-NN queries over index structures are the branch-and-bound and the incremental approaches. The branch-and-bound approach traverses the indexing tree from the root, employing at each step a heuristic to determine if the branch can be pruned from the search. One of the landmark branch-and-bound k-NN algorithm was proposed in [Roussopoulos et al. 1995], traversing an R-tree [Guttman 1984] in depth-first fashion. Regarding MAMs, the M-tree introduced a variation of the branch-and-bound approach [Ciaccia et al. 1997]. The M-tree algorithm, which was also included in the Slim-tree, maintains a global priority list containing the nodes not yet traversed, which allows proceeding the search from the node closest to the query reference. Such visiting order improves the search performance, thus this algorithm is known as the *Best-First k-NN*. The incremental approach aims at retrieving with minimal effort the $k+1$ nearest neighbor after the first k have been found. The *Incremental k-NN* algorithm [Hjaltason and Samet 2003] delays as much as possible real distance calculations, using a priority queue containing the elements not yet visited. This algorithm is optimal regarding both disk accesses and distance calculations, however, the number of queue operations is usually very high, which drastically degrades the execution performance, being much slower than the Best-Fit algorithm regarding execution time [Vieira et al. 2007].

There are a few approaches that allow posing queries combining a k-NN search with additional filters on other attributes. DBMS vendors have released modules to perform similarity queries inside their database engines, such as the Oracle interMedia [Oracle Corporation 2005] and the IBM DB2 Image, Audio and Video Extenders [IBM Corporation 2003], which include proprietary algorithms and data structures. Open-source systems adding similarity retrieval functionalities to commercial DBMS have also been developed, such as the FMI-SiR (user-defined Features, Metrics and Indexes for Similarity Retrieval) [Kaster et al. 2010], the SIREN (Similarity REtrieval ENgine) [Barioni et al. 2005] and the PostgreSQL-IE [Guliato et al. 2008] systems. Although such approaches allow combining similarity and built-in DBMS operators, the combination is restricted to conjunctions and/or disjunctions of filtering conditions, which cannot answer queries involving aggregate-based conditions.

There also are works combining a k-NN query and a filter on a traditional attribute into a single index. For instance, in [Park and Kim 2003] an incremental k-NN algorithm over the RS-tree was proposed, which is a hybrid of the R-tree and the S-tree [Deppisch 1986], based on a hierarchical signature file. However, this algorithm supports only a keyword-based equality condition involving the traditional attribute, as the signature file generated by the S-tree is targeted to keyword-based queries. A similar approach is presented in [Lee and Park 2005], which proposes another structure based on a variation of the RS-tree, called the SPY-TEC+ index. The SPY-TEC+ is tailored to high-dimensional content-based retrieval, but it is also restricted to keyword-based conditions. Another work combining similarity and keyword-based conditions is presented in [De Felipe et al. 2008]. This work proposes a method to efficiently answer top-k spatial queries with conditions based on traditional attributes using the Information Retrieval R-tree (IR^2-tree), which integrates of Information Retrieval (IR) techniques and data structures and algorithms used in spatial databases. However, this approach cannot execute k-NNq with aggregate-based conditions either.

Therefore, to the best of our knowledge, there is no work in the literature addressing k-NN queries constrained by aggregate-based conditions as those we propose in this article.

3. CONDITION-EXTENDED K-NN QUERIES

The k-Nearest Neighbor query can be extended to include additional conditions. We define in this article a variation on the k-NN query as a *condition-extended k-NN query*, represented as ${}_c k$-NNq. A ${}_c k$-NNq is an exact query that retrieves the k elements nearest to the query reference that meet a given condition – or assertion, using a well known SQL terminology. Generally speaking, a traditional k-NN query can be understood as an operation that sorts all the elements in the dataset according to their dissimilarity to the query reference and takes the first k elements in this list. Following this reasoning, a ${}_c k$-NNq can also be answered sorting the elements according to the similarity to the reference element, however, it selects the first k elements in the list that satisfy the query assertion. To formally state the meaning of the ${}_c k$-NNq, we introduce the definitions following, regarding a complex data domain $\mathbb{S}$, a set of elements $S \in \mathbb{S}$, a distance function δ defined over $\mathbb{S}$, a reference element $s_r \in \mathbb{S}$ and a condition c defined over elements in S.

Definition 3.1. A *dissimilarity-induced set* S^{δ,s_r} is a set containing the elements $s_i \in S$, $|S^{\delta,s_r}| = |S|$, such that the elements have an additional property, the *dissimilarity value* regarding s_r, defined as $\delta(s_i, s_r)$. It is said that S^{δ,s_r} is a dissimilarity set induced by S, δ and s_r.

Definition 3.2. A *k-dissimilarity-induced set* $S^{\delta,s_r}_{c,k}$ is a subset of a dissimilarity-induced set S^{δ,s_r} such that $|S^{\delta,s_r}_{c,k}| = k$ and whose elements satify the condition c.

Definition 3.3. A *minimal k-dissimilarity-induced set* $\check{S}^{\delta,s_r}_{c,k}$ is a k-dissimilarity-induced set whose sum of the dissimilarity values of its elements is minimal regarding any k-dissimilarity set induced by S, δ, s_r and c.

According to these definitions, the answer set of a ${}_c k$-NNq regarding a dataset S, a query element $s_q \in \mathbb{S}$, a distance function δ and a condition c, is the minimal k-dissimilarity-induced set $\check{S}_{k,c}^{\delta,s_q}$.

In the same way as the regular k-NNq, the ${}_c k$-NNq operation can be algebraically represented through the selection variation $\ddot{\sigma}$, but with a predicate having the form $\langle S \ {}_c k\text{NN}[\delta, k, cond] \ s_q \rangle$, where S is an attribute in the domain $\mathbb{S}$ of the input relation, $s_q \in \mathbb{S}$ is the reference element and ${}_c k\text{NN}[\delta, k, cond]$ is the ${}_c k$-NN operator, being δ the employed distance function, k the number of neighbors and *cond* the condition modifying the nearest neighbor search. The condition of a ${}_c k$-NNq is expressed using attributes in the input relation and it can be null, a tuple-based condition or an aggregate-based condition. If *cond* is null, the ${}_c k$-NNq is a regular k-NNq. A *Tuple-based condition*, or *t-cond*, is a condition that can be verified analyzing just one tuple at a time. For instance, considering a relation $Exams(imageId, studyId, patientName, patientAge, image)$ storing the tuple of medical exam images, as well as its identification, the identification of the exam which it belongs and the name and age of the patient, the query $\ddot{\sigma}_{image \ {}_c k\text{NN}[\delta,k,patientAge \geq 65] \ queryImage}(Exams)$ uses a ${}_c k$-NNq with a *t-cond*, as the condition $\langle patientAge \geq 65 \rangle$ is able to be checked just accessing each single tuple. An *Aggregate-based condition*, or *a-cond*, is a condition employing an aggregation function over the query result. This means that the condition has to be satisfied regarding the subset of elements that compose the result as a whole, as aggregation functions are table-based, i.e. they inherently depends on the whole input dataset to be computed.

A ${}_c k$-NNq whose assertion includes an aggregate-based condition cannot be expressed neither as a conjunction nor as a disjunction of tuple-based conditions. Therefore, it cannot be expressed using a single query expression, and thus it cannot be optimized in a DBMS. As evaluating similarity-based operators are computationally costly processes, it is worth defining efficient algorithms to execute ${}_c k$-NN queries with aggregate-based conditions, such that this operation can be included in a query expression together with other query operators and efficiently executed by the DBMS. In this article, we explore ${}_c k$-NN queries constrained by one type of aggregate-based condition, called the counting aggregate-based condition, presented in the next section.

4. COUNTING AGGREGATE-BASED CONDITIONS FOR EXTENDED K-NN QUERIES

We define the *Counting Aggregate-based similarity condition*, or *ca-cond*, as an aggregate-based condition of a ${}_c k$-NN query that employs a filtered count aggregate function, stated as follows:

$$\text{COUNT}(aggAttr, t\text{-}cond) \ \Theta \ c$$

where *aggAttr* is the attribute employed to count elements, *t-cond* is a tuple-based condition to filter the elements that are considered in the COUNT aggregation function, Θ is one of $\{<, \leq, \geq, >\}$ and $c \geq 0$ is an integer value. Therefore, a *ca-cond* defines that the number of elements in the ${}_c k$-NNq result that satisfy the provided *t-cond*, and whose values of *aggAttr* are not null, have to be Θ c. Alike the regular COUNT aggregation function of database systems, the parameter *aggAttr* can be replaced by an $*$, indicating counting tuples, instead of non-null attribute values.

This construction allows defining several useful queries, including those presented in the Introduction. For example, Query 1 ("*return the* 20 *cities that are the closest ones to Bristol city, USA, such that the result includes at least* 5 *cities having a population of* 100,000 *or more people*") can be posed as follows.

$$\ddot{\sigma}_{coord \ {}_c k\text{NN}[L_2,20,\text{COUNT}(*,pop \geq 100{,}000) \geq 5] \ BristolCoord}(Cities)$$

Similarly, Query 2 ("*return the* 10 *hotels that are the closest to the conference meeting place, such that at most* 2 *of them are* 10 *miles or more far from the airport*") can be stated as a condition-extended k-NN query as follows, considering the table $Hotels(name, amenities, roomRates, coord)$:

$$\ddot{\sigma}_{coord \ {}_c k\text{NN}[L_2,10,\text{COUNT}(*,L_2(coord,airportCoord) \geq 10 \text{ miles}) \leq 2] \ meetingCoord}(Hotels)$$

where *airportCoord* and *meetingCoord* are constants representing the geographical coordinates of the respective places and *coord* is an attribute of the input table.

A variation of the counting aggregate-based condition consists in adding the modifier DISTINCT to the COUNT aggregate function. The DISTINCT modifier makes the query to return at least/most c elements satisfying *t-cond* having distinct values for the attribute $aggAttr$. This variation allows posing queries such as Query 3 ("*return the* 10 *medical images are the most similar to the query image* i_q*, such that every image returned is of a distinct exam study*"), over the relation $Exams$, presented in the previous section, as follows:

$$\ddot{\sigma}_{image\ {}_c kNN[L_2,10,\text{COUNT(DISTINCT}(studyId))\geq 10]\ i_q}(Exams)$$

Notice that this aggregate-based condition does not employ a filtering tuple-based condition (*t-cond*), thus every element is considered in the aggregate function. Being the attribute *studyId* the study identifier, the query will not return more than one image of a same study, as the value of c is equal to k. One example of query using the COUNT(DISTINCT) function and a *t-cond* is shown as Query 4.

Query 4: *Return the* 10 *images that are the most similar to the reference image* i_q*, such that at least* 8 *of them are from distinct exams/studies of senior citizens (*65 *years old or more) and the remainder images can be of younger patients.*

Query 4 can be written using the ${}_ck$-NNq operation as follows:

$$\ddot{\sigma}_{image\ {}_c kNN[L_2,10,\text{COUNT(DISTINCT}(studyId,patientAge\geq 65))\geq 8]\ i_q}(Exams)$$

It is worth mentioning that this query is different from the ${}_ck$-NN query with tuple-based condition presented in Section 3, because it looks for images from distinct exams as well as it allows returning images of younger patients, even being more than one in the same study. The reason for allowing returning more than one image per study from younger patient exams is that the *t-cond* filter the elements that should be considered in the aggregate function, thus only senior citizens' exam images are counted and checked for distinctness. The meaning behind this condition in this query is that if there are images of younger patients that are more similar to the reference one than images of senior citizens, it is desired to see them, regardless of they being from the same exam or not.

The behavior of the COUNT(DISTINCT) aggregate-based condition can lead to a misleading on the meaning of the inverse, or negated form of the condition. Regarding the COUNT condition the following equivalence rule is valid:

$$\text{COUNT}(aggAttr, \textit{t-cond}) \geq c \Leftrightarrow \text{COUNT}(aggAttr, \neg \textit{t-cond}) < (k - c + 1)$$

Therefore, for instance, Query 1 could be alternatively written using the following condition:

$$\ddot{\sigma}_{coord\ {}_c kNN[L_2,20,\text{COUNT}(*,pop<100{,}000)<16]\ BristolCoord}(Cities)$$

However, the same notion is not valid when employing the DISTINCT modifier. That is:

$$\text{COUNT(DISTINCT}(aggAttr, \textit{t-cond})) \geq c \not\Leftrightarrow \text{COUNT(DISTINCT}(aggAttr, \neg \textit{t-cond})) < (k - c + 1)$$

as the left side of the rule allows returning several elements that do not satisfy the *t-cond* and have the same $aggAttr$ value, while in the right side of the rule these elements (i.e. the elements not satisfying *t-cond*, and consequently satisfying $\neg$*t-cond*) are checked for distinctness regarding $aggAttr$, generating different results.

Figure 1 illustrates results of a ${}_ck$-NN query using $k = 10$, regarding variations of *ca-cond*. The region shown in the figures is the neighborhood of the Bristol city, in the Virginia state, USA, near to

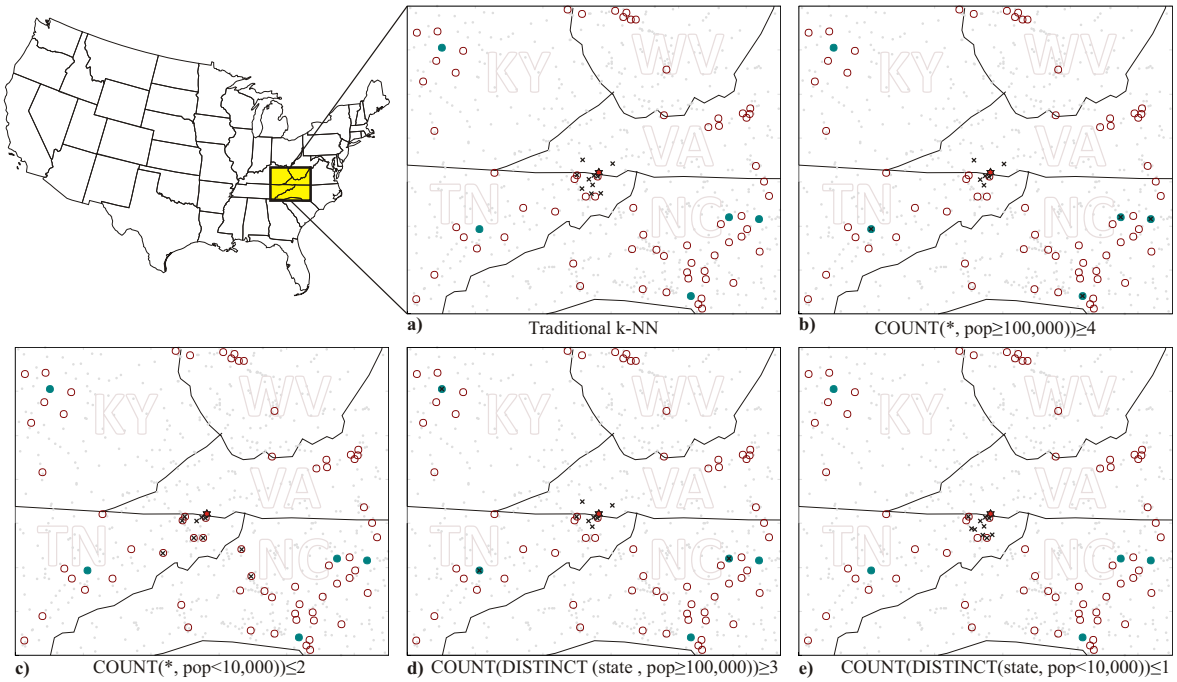

Fig. 1. Comparation of the results of a k-NNq and of ${}_c k$-NN queries with variations of counting aggregate-based conditions, defined below each figure.

the division with the Tennessee state. The query reference is the Bristol city, which is represented by the red star, and was previously removed from the dataset to make the examples clearer. The solid circles are cities having population of 100,000 or more people, the circles are cities/counties having population between 10,000 and 99,999 people and the small gray points are cities/counties having less than 10,000 people. The cities that compose the query result are shown as small crosses. Figure 1a shows the result of a regular k-NNq and figures 1b-1e show results of ${}_c k$-NNq constrained by the *ca-cond* indicated above each figure. The *ca-cond* of Figure 1b included 4 cities with 100,000 or more people that were not in the result of the k-NNq of Figure 1a. Figure 1c shows that the provided *ca-cond* included only the 2 closest cities/counties that have population of less than 10,000 people. Figure 1d selected cities having more than 10,000 people of at least three distinct states (notice the state divisions). Finally, Figure 1e included in the result only cities having less than 10,000 people from the North Carolina state, as the *ca-cond* defines that they should be of the same state (≤ 1) and the nearest cities having population of less than 10,000 are in the North Carolina. Notice that two cities with population between 10,000 and 100,000 inhabitants are in the result, which, fortuitously, also are in South Carolina. However, the result could include cities with 10,000 or more inhabitants from other states, if they were close enough to the query reference.

5. ALGORITHMS FOR COUNTING AGGREGATE-BASED CONSTRAINED K-NN QUERIES

This section presents the algorithms developed to execute k-NN queries with counting aggregate-based conditions, employing sequential and indexed scans. The algorithms assume that each complex element is stored in a table containing the feature vector(s) as well as other attributes describing the element. The data structure employed to generate the result is a "three-way" priority list ordered by the dissimilarity value of the elements. This structure is composed of three sub-lists: the first storing the elements satisfying the counting condition, the second storing the elements that do not satisfy the counting condition and the third merging all elements in the result.

5.1 Sequential Scan Algorithms

Algoritm 1 is the proposed sequential scan solution for executing ${}_ck$-NN queries with **counting** aggregate-based conditions whose count value is **greater than or equal to** the provided value, such as the *ca-cond* aforecited for representing Query 1. This algorithm starts initializing the result data structure and the query radius, which is used to dynamically prune the search space (lines 1-2). For every tuple in the table data pages, the distance between the tuple and the query element is evaluated, and if the distance is smaller than the dynamic radius, the tuple is qualified to be included in the result regarding the dissimilarity (lines 3-6). Thereafter the algorithm checks if the tuple satisfies the provided *t-cond* and if the value of the aggregation attribute is not null, to decide whether the tuple must be considered in the counting (line 7). If the tuple fulfill all these requirements, it is included in the result in the satisfying elements sub-list, and, if this insertion makes the result to have more than k elements, the farthest element is cut out, maintaining exactly k elements in the result, as informed in the function parameter (lines 8-9). Otherwise, tuple is a non-satisfying element. In this case, it is verified whether there is room in the result for storing a non-satisfying element without breaking the counting condition, which states that at least c elements have to satisfy the *t-cond*, and therefore the result can have at most $k - c$ non-satisfying elements (line 11). If there is, the tuple is added in the non-satisfying sub-list (line 12). If after this operation the result has more than k elements it means that there were more than c elements in the satisfying sub-list, so the farthest element can be safely dropped, regardless of being a satisfying or non-satisfying element (line 13). If there is no room for additional non-satisfying elements (i.e. the result already stores $k - c$ non-satifying elements), the current tuple will only be included in the result if it is closer to the query element than the farthest non-satisfying element (line 15). If the tuple is closer than the farthest, they are exchanged in the result by adding the tuple in the non-satisfying sub-list and cutting out the farthest element in this sub-list (lines 16-17). Finally, the algorithm checks if the result has k elements and, in this case, the dynamic radius is updated with the maximum distance in the result, which is the distance between the farthest element in the result and the query element (lines 21-22).

Algorithm 1: ${}_ck$-NN_CountAggregate_GreaterThanOrEqual

Input: table, $\delta, k, s_q, aggAttr, t\text{-}cond, c$
Output: result

```
1  result ← ∅;
2  maxDist ← ∞;
3  foreach dataPage in table do
4      foreach tuple in dataPage do
5          distance ← δ(tuple.s, s_q);
6          if distance < maxDist then
7              if tuple satisfies t-cond and tuple.aggAttr ≠ NULL then
                   // tuple is added as a satisfying element
8                  result.addSatisfyElement(tuple, distance);
9                  if result.getNumElements() > k then result.cutElements(k);
10             else
                   // tuple is a non-satisfying element
11                 if result.getNumNotSatisfyElements() < (k − c) then
12                     result.addNotSatisfyElement(tuple, distance);
13                     if result.getNumElements() > k then result.cutElements(k);
14                 else
15                     if distance < result.getMaxDistanceNotSatisfy() then
16                         result.addNotSatisfyElement(tuple, distance);
17                         result.cutNotSatisfyElements(k − c);
18             if result.getNumElements() ≥ k then
19                 maxDist ← result.getMaxDistance();
```

The algorithm to execute ${}_ck$-NN queries with aggregate-based conditions whose **count** value is **less than or equal to** the provided value are alike to Algorithm 1. However, the less than or equal to algorithm inverts the treatment of satisfying and non-satisfying elements, as, in this case, the number of satisfying elements is which have to be limited in the result, while in the greater than or equal to approach the number of non-satisfying elements is limited.

Algorithm 2 answers ${}_ck$-NN queries with **counting distinct** aggregate-based conditions, such that the count is **greater than or equal to** the given value. The first seven lines of this algorithm are exactly equal to the respective ones in the previous algorithm. In line 8, the algorithm scans the result looking for a duplicate satisfying elements regarding *aggAttr*. If a duplicate is found, the algorithm tests if the current tuple is closer to the query reference than the duplicate (lines 9-10). If so, the tuple is included in the satisfying sub-list, replacing the duplicate (line 11). Thereafter, it is checked if the duplicate should be maintained in the result, now as a non-satisfying element. The algorithm first verifies if there is room for additional non-satisfying elements and, in this case, moves the duplicate to the non-satisfying sub-list (lines 12-13). If after this operation the result exceeds k elements, the farthest element is cut out, as it is known that, in this point of the algorithm, the result contains more than c satisfying elements (line 14). If the number of non-satisfying elements is in the borderline, the duplicate is only maintained if it is closer than the farthest non-satisfying element (line 15). If it is in the border, the duplicate is moved to the non-satisfying sub-list and the farthest non-satisfying element is cut out (lines 16-17). If the duplicate does not meet any of the two alternatives, it is removed from the result. In the case of the tuple (satisfying *t-cond*) being farther than the duplicate, the algorithm checks if the tuple should be added to the result as a non-satisfying element (lines 22-27), following a logic similar to that used earlier to verify if the farther duplicate should be maintained in the result. Lines 30-31 handles the tuples satisfying the element-based condition that do not have duplicates in the result. In this case, the tuple is added to the result, and, if the result exceeds k elements, the last one is cut out, as in this point it is guaranteed that the number of distinct satisfying elements in the result is greater than c. Tuples that do not satisfy the provided *t-cond* follow the aforementioned procedure regarding non-satisfying elements. Finally, if the result holds k elements, the dynamic radius is updated (lines 37-38) and the search continues until the final query result is obtained.

The algorithm to execute ${}_ck$-NN queries with **counting distinct** aggregate-based conditions whose number of satisfying elements is **less than or equal to** the provided value has a trickier treatment of satisfying elements, requiring maintaining additional candidate lists. Due to space limitations, this algorithm is not shown in this article. However, we present a quick the intuition. A ${}_ck$-NNq with a $\text{COUNT}(\text{DISTINCT}(aggAttr, t\text{-}cond)) \leq c$ condition, such that $c < k$, allows more than one satisfying tuple having the same value for *aggAttr* in the result, where only one of them is considered in the aggregate function. During the execution of the query, if the number of (distinct) satisfying elements in the result is in the borderline and the algorithm reachs another satisfying tuple that is distinct from the remainder satisfying tuples in the result, it should replace another satisfying one. However, if the replaced tuple have duplicates, they must also be removed from the result. When this occurs, previous duplicate satisfying tuples that were not included in the result may now be included. Therefore, it is necessary to maintain additional duplicate satisfying tuples to complete the result when such situation occurs.

5.2 Algorithms in the Slim-tree Metric Access Method

While the traditional k-NN algorithm needs only the feature vectors and a distance function to compare them, the counting aggregate-based algorithms also demand additional attributes to be executed. It is not difficult to realize that this extra information can be explored to develop new indexing structures. However, this option is out of the scope of this article, thus we choose developing algorithms for executing ${}_ck$-NN queries with counting aggregate-based conditions for the Slim-tree, which is a consolidated structure for metric and complex data.

Algorithm 2: ${}_ck$-NN_CountDistinctAggregate_GreaterThanOrEqual

```
   Input: table, δ, k, s_q, aggAttr, t-cond, c
   Output: result
1  result ← ∅;
2  maxDist ← ∞;
3  foreach dataPage in table do
4      foreach tuple in dataPage do
5          distance ← δ(tuple.s, s_q);
6          if distance < maxDist then
7              if tuple satisfies t-cond and tuple.aggAttr ≠ NULL then
8                  duplicate ← result.lookForSatisfyingDuplicate(tuple.aggAttr);
9                  if duplicate ≠ NULL then
10                     if δ(tuple.s, s_q) < δ(duplicate.s, s_q) then
11                         result.addSatisfyElement(tuple, distance);
                           // Check whether duplicate is maintained, now as non-satisfying
12                         if result.getNumNotSatisfyElements() < (k − c) then
13                             result.toNotSatisfy(duplicate);
14                             if result.getNumElements() > k then result.cutElements(k);
15                         else if δ(duplicate.s, s_q) < result.getMaxDistanceNotSatisfy() then
16                             result.toNotSatisfy(duplicate);
17                             result.cutNotSatisfyElements(k − c);
18                         else
19                             removeSatisfyElement(duplicate);
20                     else
                           // Treat tuple as a non-satisfying element
21                         if result.getNumNotSatisfyElements() < (k − c) then
22                             result.addNotSatisfyElement(tuple, distance);
23                             if result.getNumElements() > k then result.cutElements(k);
24                         else if δ(tuple.s, s_q) < result.getMaxDistanceNotSatisfy() then
25                             result.addNotSatisfyElement(tuple, distance);
26                             result.cutNotSatisfyElements(k − c);
27                 else
28                     result.addSatisfyElement(tuple, distance);
29                     if result.getNumElements() > k then result.cutElements(k);
30             else
                   // Treat tuple as a non-satisfying element
31                 ...
32             if result.getNumElements() ≥ k then
33                 maxDist ← result.getMaxDistance();
```

We implemented in the Slim-tree two variations of each algorithm presented in the previous section, according to the way employed to access the additional attributes. Both of them follow the logic of the respective linear algorithms with regard to the management of the result data structure. However, they also include the logic to prune subtrees and to avoid unnecessary distance calculations using the triangle inequality property and use the the Best-First k-NN strategy to define the node visiting order.

The first variation, called the *Table-Slim* algorithms, uses a regular Slim-tree, storing only the feature vector and the row identifier (*rowId*) of each indexed element. The algorithms check if the visited elements are qualified regarding the distance to the query reference inside the tree. However, to check the conditions based on external attributes, the algorithms retrieve the tuples from the table, thus, accessing data pages during the search. Although this variation can demand a huge amount of data page accesses, it is a generic alternative that allows employing any external attribute together with a Slim-tree and, depending on the query, it can be much faster than the sequential scan.

The second variation, called the *Covering-Slim* algorithms, defines the Slim-tree as a covering index. This means that the Slim-tree stores additional attributes associated to the indexed element, besides the feature vector and the *rowId*. Thus, if a Covering-Slim index including the external attributes used in the ${}_ck$-NNq condition is available, it is not necessary to access data pages during the search, as all data required to execute the search is in the index.

6. EXPERIMENTS

6.1 Datasets and Experiment Description

We have executed experiments evaluating the proposed algorithms over geographic and image datasets. This section shows results achieved over two of them. The first dataset, called the USCities dataset, contains the name, state and geographic coordinates of 25374 cities/counties in the United States together with 97 associated (numeric) demographic attributes. This dataset was obtained combining demographic data from the year 2000 census of the United States and the coordinates from the Tiger system, both provided from the U.S. Census Bureau. Regarding this dataset, we executed queries similar to Query 1, with and without the DISTINCT modifier. The performance of a ${}_ck$-NNq is affected by the number of neighbors (k), by the value of c and by the selectivity of the *t-cond* provided. It is easy to notice that a greater than or equal to c counting aggregate condition with a *t-cond* that is satisfied by only 1% of the input table tends to be costlier than a query having a *t-cond* that selects 20% of the table. Therefore, to allow evaluating the impact of these parameters, the queries in this test were stated as follows: **Query 5:** "*return the k cities that are the closest to the reference city, such that the result includes at least c cities having population greater than or equal to x*"; and **Query 6:** "*return the k cities that are the closest to the reference city, such that the result includes at least c cities, of distinct states, having population greater than or equal to x*". We executed three rounds of 500 queries, using the L_2 distance for the coordinates, each query with a distinct city reference, being: 1^{st} round) varying c from 1 to 100 for Query 5 and from 1 to 10 for Query 6 and, for both these queries, fixing $k = 100$ and the *t-cond* selectivity in 5% (using $x = 29674$); 2^{nd} round) varying k from 20 to 400 and fixing $c = 20$ for Query 5 and $c = 5$ for Query 6 and $x = 29674$ for both queries; and 3^{rd} round) varying the *t-cond* selectivity from 1% to 50% and fixing $c = 20$ for Query 5 and $c = 5$ for Query 6 and $k = 100$ for both queries.

The second dataset, called the DICOM_HC dataset, was obtained from a collection of 200,000 DICOM images from the Ribeirão Preto Medical School Clinics Hospital of the University of São Paulo (HCFMRP-USP). From each image, we extracted a 256-bin grayscale normalized histogram and 13 attributes from the image header, including exam description, patient age and some patient information. The queries executed over this dataset, comparing the histograms using the L_1 distance, were: **Query 7:** "*Return the k images that are the most similar to the given reference image, such that at least c of them are from patients whose age is x years old or more*"; and **Query 8:** "*Return the k images that are the most similar to the given reference image, such that at least c of them are from distinct exams/studies of patients whose age is x years old or more and the remainder images can be of younger patients*". With regard to this dataset, we also performed three rounds of queries with parameters similar to the USCities dataset, but taking the average of 50 queries with random reference images. Moreover, regarding Query 8, when the c parameter was varied, it was from 1 to 35, and when the other parameters were varied, c was fixed in 15.

The experiments were run on a dedicated machine equipped with an Intel Core 2 Quad 2.83GHz processor, 4GB of RAM and a SATA2 HD of 2TB and 7,200RPM, running the Kubuntu GNU/Linux 10.04 64bit. The algorithms were implemented in C++, using the Arboretum library[1], and compiled with GCC 4.4.3 for GNU/Linux.

[1]http://gbdi.icmc.usp.br/arboretum

6.2 Results and Discussion

The figures 2 and 3 show, respectively, the results of answering Query 5 over the USCities dataset and Query 7 over the DICOM_HC dataset. The graphs in the first column of the figures show the behavior of the developed algorithms with regard to the variation of the c parameter, while the graphs of the second and of the third columns show, respectively, the algorithms' behavior regarding the variation of k and of the *t-cond* selectivity, being the graphs of the third column always presented in log scale. The graphs on top of the figures show the wall clock time demanded to execute the algorithms over the datasets according the variation on the parameters, summarizing the behavior of the other observed variables: average number of distance calculations, average number of comparisons involving the traditional/external attributes, average number of disk accesses (of data and index pages) and average number of operations in the three-way priority list storing the result.

With regard to the USCities dataset, analyzing the graph of Fig. 2a, it can be seen that both the Sequential and the Covering-Slim algorithms presented an almost constant behavior regarding the variation of c, being the Covering-Slim up to more than 4 times faster than the Sequential. Observing the other graphs in this column, it can be seen that the variables that presented an increasing behavior, which were the number of comparisons (Fig. 2d) for the Sequential algorithm and the number of distance calculations (Fig. 2d) and the number of comparisons (Fig. 2g) for the Covering-Slim algorithm, were compensated by the decreasing behavior of the number of list operations (Fig. 2m). On the other hand, the Table-Slim algorithm presented a linear increasing behavior regarding time, performing poorly than the Covering-Slim for every value of c and poorly than the Sequential for $c > 30$. Although for most variables the Table-Slim performed slightly better than the Covering-Slim, the Table-Slim was very costly regarding disk accesses (Fig 2j), impacting in the execution time. Fig. 2b shows that the three algorithms presented a linear curve with the increasing of k. Again, the Covering-Slim was the fastest, being up to 6 times faster than the Sequential and up to 5 times faster than the Table-Slim. However, in this case, the Table-Slim was around 20% faster than the Sequential, because the larger number of disk accesses required by the Table-Slim (Fig. 2k) was compensated by its much better performance in the remainder variables comparing to the Sequential (figures 2e, 2h and 2n). The graph of Fig. 2c shows that only the Table-Slim presented acute degradation in the execution time for high *t-cond* selectivities. Again, such poor performance was mainly due to the large amount of disk accesses (Fig 2l). Both the Covering-Slim and the Sequential presented an almost constant behavior regarding the *t-cond* selectivity, having a slightly increase in the execution time for selectivities greater than 97%, due to the comparisons (Fig. 2i) in the Sequential and the distance calculations (Fig. 2f) and comparisons (Fig. 2i), as for all algorithms the number of list operations was almost constant (Fig 2o). The Covering-Slim was the best regarding time, being around 4 times faster than the Sequential and up to 6.5 times faster than the Table-Slim. The Table-Slim performed better than the Sequential for selectivities smaller than 97%, being up to 2.5 times faster.

With regard to the execution of Query 7 over the DICOM_HC dataset, Fig. 3a shows that the Sequential algorithm presented the poorest performance, with a constant execution time. Differently from the USCities dataset, querying the DICOM_HC dataset using Table-Slim presented a sub-linear behavior, being faster than the Sequential Scan for every value for c. Covering-Slim presented an interesting execution time curve, having the maximum value in the middle of the curve. The reduction occurred after the middle value was due the reduction in the number of list operations (Fig. 3m), which in this dataset is more costly than in the previous one, as the feature vector is much larger. The Covering-Slim was up to 29% faster than the Table-Slim and 33% faster than the Sequential, while the Table-Slim was up to 23% faster than the Sequential. Again, the Table-Slim and the Covering-Slim performed very close regarding distance calculations (Fig. 3d) and comparisons (Fig. 3g), being several times faster than the Sequential Scan. Nevertheless, observing Fig. 3j it can be noticed that the Table-Slim demanded less disk accesses than the Sequential when $c < 70$ and just up to 20% more accesses in the worst case. According to the variation of k, Fig. 3b shows that the execution time of both the Sequential and the Table-Slim presented an almost constant behavior, while

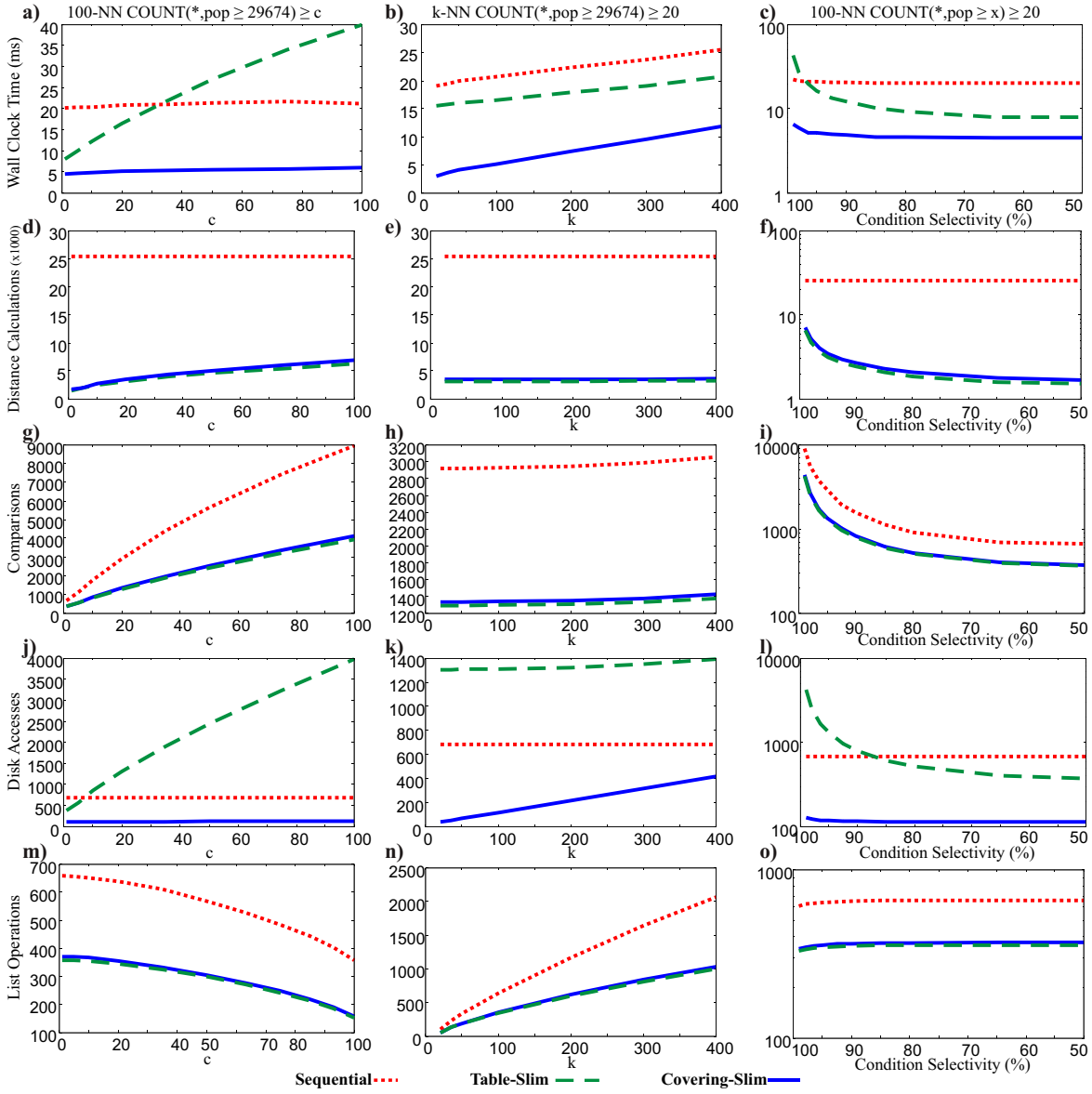

Fig. 2. Results of executing Query 5 over the USCities dataset comparing the behavior of the Sequential, Table-Slim and Covering-Slim algorithms with the variation of c (left column), k (middle column) and *t-cond* selectivity (right).

the Covering-Slim presented a sub-linear curve. However, the Covering-Slim was the more efficient, being up to 67% faster than the Sequential and up to 45% faster than the Table-Slim. The Table-Slim was up to 24% faster than the Sequential, having performed between the Covering-Slim and the Sequential in every observed variable, being better than the Sequential even regarding disk accesses (figures 3e, 3h, 3k and 3n). With regard to the *t-cond* selectivity, Fig. 3c shows that the execution time of both indexed approaches was smaller than the Sequential algorithm, for every selectivity that was tested (from 50% to 99%). The Covering-Slim was up to 33% faster than the Sequential and up to 20% faster than the Table-Slim, which was up to 25% faster than the Sequential. It can also be seen that the Sequential presented a constant behavior regarding execution time, which was the same behavior of the other variables, excepting the number of comparisons (Fig. 3i). On the other hand, the indexed algorithms presented a constant behavior only regading the number of list operations

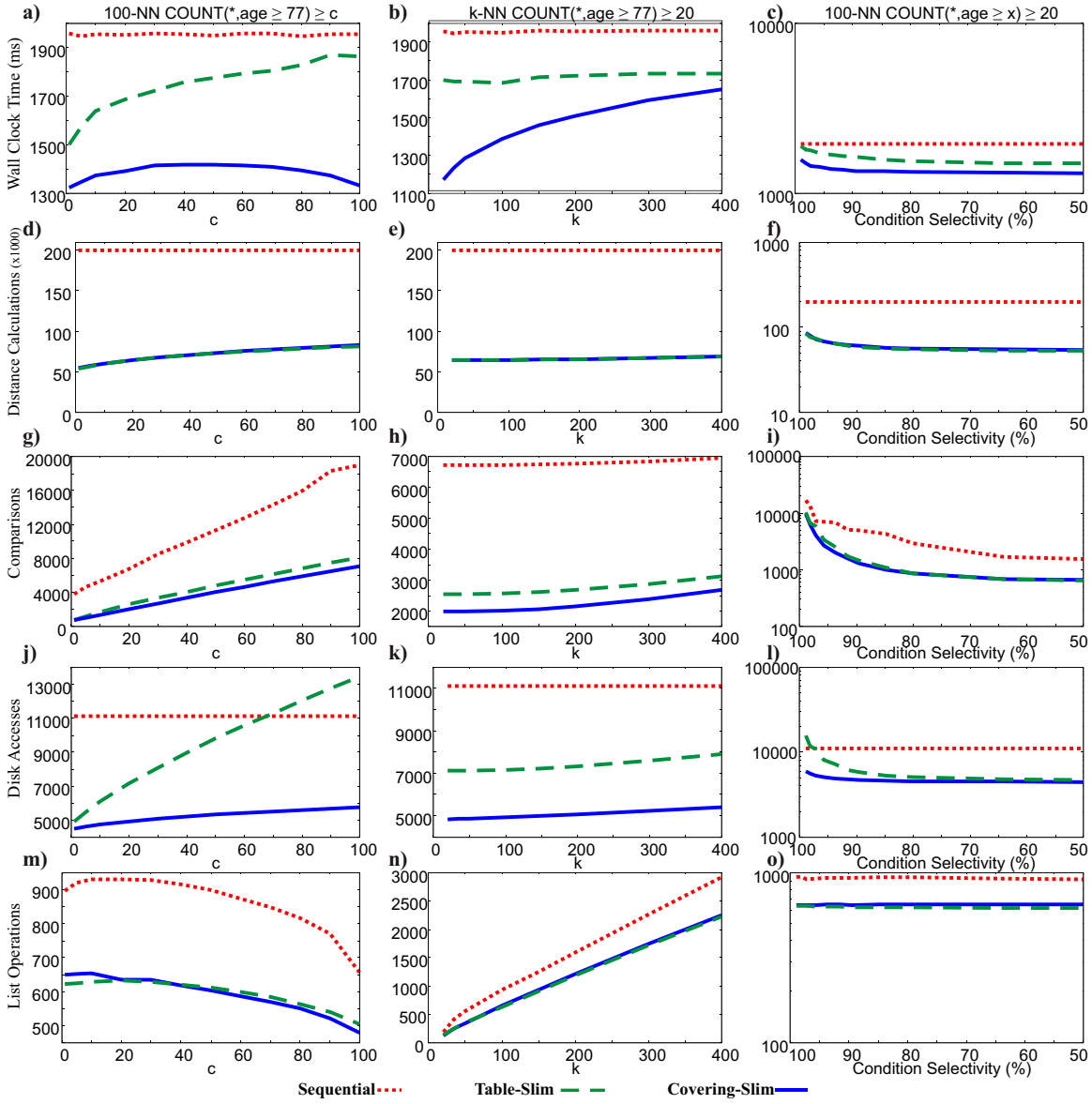

Fig. 3. Results of executing Query 7 over the DICOM_HC dataset comparing the behavior of the developed algorithms according to the parameters c (left column), k (middle column) and *t-cond* selectivity (right column).

(Fig. 3o), having the performance degraded considering high selectivities for the other variables. The indexed algorithms produced similar curves regarding the number of distance calculations (Fig 3f) and the number of comparisons (Fig 3i), however the Table-Slim experienced a sharp increase in the number of disk accesses for selectivites greater than 90%, while the Covering-Slim presented a subtle increase for selectivities greater than 97% (Fig. 3l).

With regard to ${}_c k$-NN queries with *counting distinct* aggregate-based conditions, due to space limitations, this article only shows the algorithms' execution time behavior, summarizing the overall performance. The first line of graphs in Fig. 4 show the results of executing Query 6 over the USCities dataset and the second line presents the results of executing Query 8 over the DICOM_HC dataset. Considering the USCities dataset, the behavior of the three algorithms regarding the variation of c

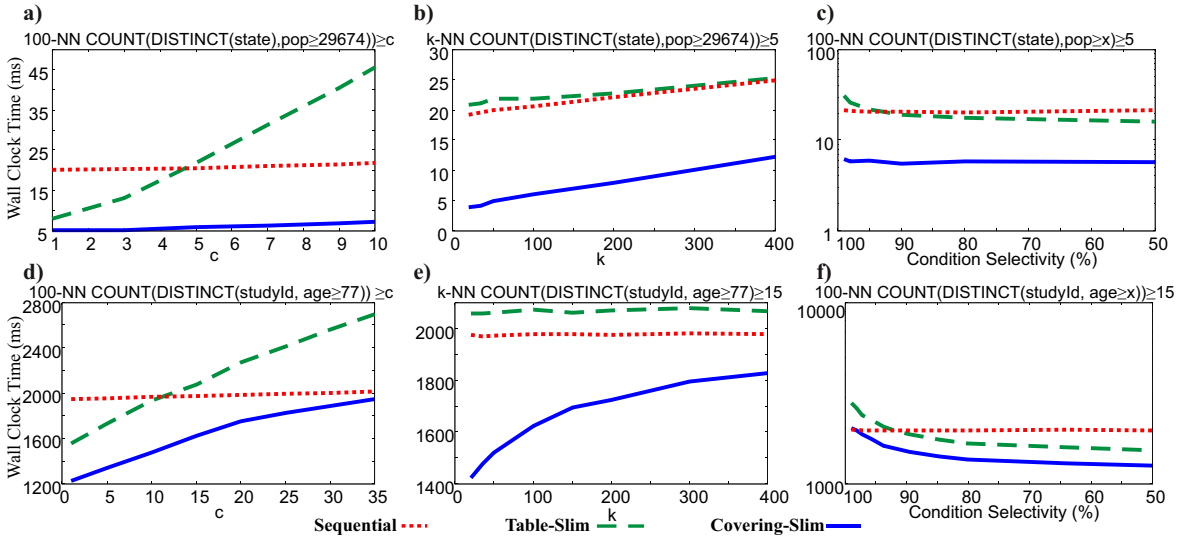

Fig. 4. Results of experiments evaluating the counting distinct aggregate-based algorithms over the datasets USCities (top) and DICOM_HC (bottom), regarding varying k (left column), c (middle column) and *t-cond* selectivity (right).

in Query 6 (Fig. 4a) was similar to the behavior of Query 5 (compare with Fig. 2a). On the other hand, varying k while the algorithms answer Query 6 (Fig. 4b) lead to the linear behavior of answering Query 5 and the Covering-Slim presented similar gains over the Sequential, the Table-Slim was slightly slower than the Sequential in Query 6, mainly because the number of disk accesses of the Table-Slim increased. A similar behavior can be noticed according to the variation of the *t-cond* selectivity, shown in Fig. 4c. The performance of the Table-Slim algorithm degraded due to the increase in the number of disk accesses, being only up to 25% faster than the Sequential (it was up to 2.5 times faster than the Sequential in Query 5). The Covering-Slim did not experienced a substantial increment in the observed variables, being around 3.5 times faster than the Sequential. Considering the DICOM_HC dataset, both indexed algorithms presented a sub-linear behavior with the increase of c, while the execution time of the Sequential was invariant to this aspect (Fig. 4d). The Covering-Slim was the fastest, but being very close to the Sequential with high values for c, and the Table-Slim was only faster than the Sequential for $c \leq 10$. When varying k, Fig. 4e shows that the execution time of the Covering-Slim incresed sub-linearly, nevertheless outperforming the Sequential in up to 30%. On the other hand, the Table-Slim had a poorer performance than the Sequential, being 4.5% slower in the average, which was promoted again by the excessive number of disk accesses. Finally, with respect to the *t-cond* variation in Query 8, the behavior of the algorithms was similar to such variation in Query 7. However, in Query 8 the impact in the performace of the indexed algorithms regarding high seletivities was more visible. The Covering-Slim tied with the Sequential for the selectivity of 99%, but it was up to 37% faster than the Sequential concerning lower selectivies. The Table-Slim was slower than the Sequential for *t-cond* selectivies greater than 92% and up to 23% faster with respect to lower selectivities. When compared to the Covering-Slim, the Table-Slim was always slower, demanding up to 25% more time to execute.

Observing the results achieved, we can conclude that the indexed approaches improved the performance to a great extent, specially the Covering-Slim algorithm, which proved to be adequate to execute the queries proposed in this article.

7. CONCLUSIONS AND FUTURE WORK

In this article, we proposed an extension to the k-NN similarity query, called the condition-extended k-NN query ($_c k$-NNq). The $_c k$-NN query allows providing an assertion over attributes associated to the complex data modifying the search, which can be either based on a single tuple, a tuple-based condition, or based on the query result set, which is an aggregate-based condition. In particular, this work addressed counting aggregate-based conditions (*ca-cond*) for $_c k$-NN queries, which are conditions based on a filtered count aggregate function that the query result must satisfy. We defined the *ca-cond* and presented several examples of queries that can be answered employing it in a $_c k$-NNq. We also presented the main algorithms to solve $_c k$-NN queries with *ca-cond*, and developed three variations of each algorithm. The first variation is based on a sequential scan and the two other variations were developed under the MAM Slim-tree. The experiments presented in the article showed that the performance of the search can be largely boosted using an appropriate access method. Future works include to address other types of aggregate-based conditions to $_c k$-NN queries and explore their peculiarities to develop new algorithms and indexing structures to efficiently execute them.

REFERENCES

Barioni, M. C. N., Kaster, D. S., Razente, H. L., Traina, A. J. M., and Traina Jr, C. Querying Multimedia Data by Similarity in Relational DBMS. In L. Yan and Z. Ma (Eds.), *Advanced Database Query Systems: Techniques, Applications and Technologies*. IGI Global, Hershey, Pennsylvania, USA, 2011.

Barioni, M. C. N., Razente, H. L., Traina Jr., C., and Traina, A. J. M. Querying complex objects by similarity in SQL. In *Proceedings of the Brazilian Symposium on Databases*. Uberlândia, Brazil, pp. 130–144, 2005.

Ciaccia, P., Patella, M., and Zezula, P. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the International Conference on Very Large Data Bases*. Athens, Greece, pp. 426–435, 1997.

De Felipe, I., Hristidis, V., and Rishe, N. Keyword search on spatial databases. In *Proceedings of the IEEE International Conference on Data Engineering*. Washington, DC, USA, pp. 656–665, 2008.

Deppisch, U. S-tree: a dynamic balanced signature index for office retrieval. In *Proceedings of the International ACM SIGIR Conference on Research & Development of Information Retrieval*. Pisa, Italy, pp. 77–87, 1986.

Guliato, D., Melo, E. V., Rangayyan, R. M., and Soares, R. C. PostgreSQL-IE: An image-handling extension for PostgreSQL. *Journal of Digital Imaging* 22 (2): 149–165, 2008.

Guttman, A. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data Conference*. Boston, Massachusetts, USA, pp. 47–57, 1984.

Hjaltason, G. R. and Samet, H. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems* 21 (4): 517 – 580, 2003.

IBM Corporation. *Image, audio, and video extenders administration and programming guide*, 2003. DB2 UDB V.8.

Kaster, D. S., Bugatti, P. H., Traina, A. J. M., and Traina Jr., C. FMI-SiR: A flexible and efficient module for similarity searching on Oracle database. *Journal of Information and Data Management* 1 (2): 229–244, 2010.

Lee, D. and Park, D. An Efficient Incremental Nearest Neighbor Algorithm for Processing k-Nearest Neighbor Queries with Visal and Semantic Predicates in Multimedia Information Retrieval System. In G. Lee, A. Yamada, H. Meng, and S. Myaeng (Eds.), *Information Retrieval Technology - AIRS'05*. Lecture Notes in Computer Science, vol. 3689. Springer, pp. 653–658, 2005.

Oracle Corporation. *Oracle interMedia User's Guide, 10g Release 2 (10.2)*. Oracle, 2005.

Park, D. and Kim, H. An enhanced technique for k-nearest neighbor queries with non-spatial selection predicates. *Multimedia Tools and Applications* 19 (1): 79–103, 2003.

Roussopoulos, N., Kelley, S., and Vincent, F. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data Conference*. San Jose, CA, USA, pp. 80–91, 1995.

Skopal, T. and Lokoč, J. New dynamic construction techniques for M-tree. *Journal of Discrete Algorithms* 7 (1): 62–77, 2009.

Traina Jr., C., Traina, A. J. M., Faloutsos, C., and Seeger, B. Fast indexing and visualization of metric datasets using Slim-trees. *IEEE Transactions on Knowledge and Data Engineering* 14 (2): 244–260, 2002.

Vieira, M. R., Traina, Caetano, J., Traina, A. J. M., Arantes, A. S., and Faloutsos, C. Estimating suitable query radii to boost k-nearest neighbor queries. In *Proceedings of the International Conference on Scientific and Statistical Databases Management*. Banff, Canada, 2007.

Zezula, P., Amato, G., Dohnal, V., and Batko, M. *Similarity Search: The Metric Space Approach*. Advances in Database Systems, vol. 32. Springer, 2006.