A Distributed Incremental Nearest Neighbor Algorithm

Fabrizio Falchi
ISTI-CNR
Pisa, Italy
fabrizio.falchi@isti.cnr.it

Fausto Rabitti
ISTI-CNR
Pisa, Italy
fausto.rabitti@isti.cnr.it

Claudio Gennaro
ISTI-CNR
Pisa, Italy
claudio.gennaro@isti.cnr.it

Pavel Zezula
Masaryk University
Brno, Czech Republic
zezula@fi.muni.cz

ABSTRACT
Searching for non-text data (e.g., images) is mostly done by means of metadata annotations or by extracting the text close to the data. However, supporting real content-based audio-visual search, based on similarity search on features, is significantly more expensive than searching for text. Moreover, the search exhibits linear scalability with respect to the data set size.

In this paper, we present a Distributed Incremental Nearest Neighbor algorithm (DINN) for finding nearest neighbor in an incremental fashion over data distributed between nodes which are able to perform a local Incremental Nearest Neighbor (local-INN). We prove that our algorithm is optimal with respect to both number of involved nodes and number of local-INN invocations. An implementation of our DINN algorithm, on a real P2P system called MCAN, was used for conducting an extensive experimental evaluation on a real-life dataset.

Categories and Subject Descriptors
H.3.3 [Information Search and Retrieval]: Search process; C.2.1 [Network Architecture and Design]: Distributed networks

General Terms
distributed systems, algorithms, performance

1. INTRODUCTION
Today, a large component of the Web content consists of non-text data, such as images, music, animations, and videos. Current search engines index Web documents by their textual content. For instance, web tools for performing image searching (such as the ones provided by Google, Yahoo!, or MSN Live Search) simply index the text near the image and the ALT attribute of the IMG tag, used to provide a description of an image. Images indexing methods based on content-based analysis or pattern matching (which for instance analyze the characteristics of images, i.e., features, such as colors and shapes) are not exploited at all. The problem is that these processes are significantly more expensive than text analysis. Nevertheless, what is more important is that the search on the level of features exhibits linear scalability with respect to the data search size, which is not acceptable for the expected dimension of the problem. The reason is that for this kind of data the appropriate search methods are based on similarity paradigm, which typically exploits range queries and nearest neighbor queries. These queries are computationally more intensive than exact match, since conventional inverted indexes used for text are not suitable for such data.

Besides multimedia information retrieval, there are other kinds of applications like bioinformatics, data mining, pattern recognition, machine learning, computer vision, that can take advantage of the similarity search paradigm. However, different applications have in general different similarity functions. A convenient way to address this problem is to formalize the similarity by the mathematical notion of the metric space. The data elements are assumed to be objects from a metric space when pairwise distances between the objects can be determined and the distance satisfies the properties of symmetry, non-negativity, identity, and triangle inequality. Our Distributed Incremental Nearest Neighbor DINN algorithm does not require the objects to be metric. We only suppose that the distance is non-negative.

P2P systems are considered today promising means to address the problems of scalability, and several scalable and distributed search structures have been proposed even for the most generic case of metric space searching (see [3] for a survey). A common characteristic of all these existing approaches is the autonomy of the peers with no need of central coordination or flooding strategies. Since there are no bottlenecks, the structures are scalable and high performance is achieved through parallel query execution on individual peers.

Since the number of closest objects is typically easier to specify than the search range, users prefer nearest neighbors queries. For example, given an image, it is easier to ask for 10 most similar ones according to an image similarity criterion than to define the similarity threshold quantified as a real number. However, nearest neighbors algorithms

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

INFOSCALE 2007, June 6-8, Suzhou, China
Copyright © 2007 ICST 978-1-59593-757-5
DOI 10.4108/infoscale.2007.196
are much more difficult to implement in P2P environments. The main reason is that traditional (optimum) approaches [7] are based on a priority queue with a ranking criterion, which sequentially decides the order of accessed data buckets. In fact, the existence of centralized entities and sequential processing are completely in contradiction with decentralization and parallelism objectives of any P2P search network. Things are further complicated by the natural necessity of some applications to retrieve the nearest neighbor in an incremental fashion. This because, the number of desired neighbors is unknown in advance. By incremental, we mean that such an algorithm computes the neighbors one by one, without the need to re-compute the query from scratch. An important example of application of Incremental Nearest Neighbor is processing of complex queries, i.e., queries that involve more than one feature, such as: find all images in database similar to the query image with respect to the color and the shape. In this situation, we do not know how many neighbors must be retrieved before one is found that satisfies the conditions. In fact, the widely used A₀ algorithm defined in [4] suppose that each single source for a specific feature is able to perform a local-INN algorithm.

This approach is the core of the European project SAPIR¹ (Search on Audio-visual content using Peer-to-peer Information Retrieval) that aims at finding new ways to analyze, index, and retrieve the tremendous amounts of speech, image, video, and music that are filling our digital universe, going beyond what the most popular engines are still doing, that is, searching using text tags that have been associated with multimedia files. SAPIR aims at breaking this technological barrier by developing a large-scale, distributed peer-to-peer architecture that will make it possible to search for audio-visual content by querying the specific characteristics (i.e. features) of the content. SAPIR’s goal is to establish a giant peer-to-peer network, where users are peers that produce audiovisual content using multiple devices and service providers are super-peers that maintain indexes and provide search capabilities.

In this paper, we present a first attempt to approach the incremental nearest neighbor problem for P2P-based systems. Our proposed solution, based on a generalization of the priority queue algorithm proposed in [7] for hierarchical centralized structures, is optimal and is not tied on a specific P2P architecture, and can be used over scalable and distributed data structures (SDDSs), P2P systems and Grid infrastructures. We implemented our algorithm, on a real P2P system called MCAN [5,6] and we conducted an extensive experimental evaluation on a real-life dataset of 1,000,000 objects.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 provides an overview of our proposed solution while the formal definition is given in Section 4. In Section 5 we report the results of an extensive experimental evaluation of the DINN over the MCAN. Conclusions and future work are discussed in Section 6. All Theorems, Corollaries and Conditions are given in Appendix.

2. RELATED WORK

In [12] an efficient algorithm to perform k-NN in a P2P system (specifically the Chord [11]), is proposed. The algorithm uses the same priority queue based approach of [7].

As far as we know, it is the first try to extend [7] to the distributed environment making use of the parallelism of the P2P network.

They define their algorithm for a hierarchical index (as in [7]). To provide distributed hashing of spatial data they use a distributed quadtree index they developed, although they say that other indices can be utilized as well (e.g., P2P-R-trees [8]). A NN query is first initiated on a single peer in the P2P network. This peer maintains the priority queue of quadtree blocks (mapping to a control point each) that are being processed for the query. To process a block, they have to contact from this query initiating peer, the peer that owns that block, i.e., the control point. Hence, in their parallel algorithm, they contact, rather than just the top entry of the priority queue, a multiple number of these peers.

3. DINN OUTLINE

The INN algorithm [7] was defined for a large class of centralized hierarchical spatial data structures. Instead our DINN algorithm is distributed and not limited to hierarchical structures. Thus it can be used over SDDSs, P2P systems and Grid infrastructures. Our algorithm is built over nodes which are able to perform locally an INN between the objects they store (this will be our Assumption 1).

In particular, we reformulate the definition of priority queue (Queue) given in [7] by considering as elements of Queue, objects and nodes (or peers). We prove that our algorithm is optimal, in terms of both number of nodes involved and local-INN invocations. The elements of Queue are ordered according to a key which is always associated with both objects and nodes. For each object, key is its distance from the query. For each node, we require the associated key to be a lower bound for the distance between the query and the next result coming from the node. While for an already involved node this lower bound can be simply the distance from the query of the last object retrieved by its local-INN, for the not yet involved nodes a naive solution could be to always use 0 as lower bound. However, this would imply all nodes to be involved for every similarity query. To avoid this, we suppose that each node is able to evaluate this lower bound for every node it knows (in P2P systems they are called neighbors).

Furthermore, in P2P systems there is no global knowledge of the network. Thus, we make the assumption (see Assumption 2) regarding the ability to find the next most promising node (considering the lower bound mentioned before). This assumption replace the consistency condition used in [7] for hierarchical data structures. We prove that our assumption can be satisfied under one of two simpler conditions (see Appendix) which are common for data structures able to perform similarity search. Moreover, systems satisfying one of these two conditions (i.e., Condition 1) as MCAN, can guarantee that next most promising node is always in Queue just collecting information about neighbors of yet involved nodes (i.e., without generating more message).

During the DINN algorithm execution, Queue contains a certain number of entries sorted in order of decreasing key. The algorithm proceeds by processing Queue from the top: if the first entry of the queue is an object x, it is the result of the DINN unless there are other nodes in the system which have not yet processed a local-INN and that are “closer” (considering the lower bound distance) to the query. In the latter case, our DINN takes the “closest” node and places

¹http://sysrun.haifa.il.ibm.com/sapir
it in the queue. In case the first entry is a node, we invoke its local-INN. The resulting object of this operation (if any) is placed in Queue and its distance from the query allows us to update the entry with a more accurate lower bound distance for the next object that would be retrieved by its local-INN.

This outlined implementation is intrinsically sequential, since a single step of the algorithm involves only the first element of Queue at a time. In the second part of the paper, we straightforwardly generalize the algorithm introducing parallelism by invoking the local-INN algorithm of more than one node simultaneously. A precise definition of the algorithms are provided in the next section.

4. THE DINN ALGORITHM

4.1 Definitions and Notation

In this subsection we provide a number of definitions and notations required to define the DINN algorithm. Let $\mathcal{N}$ be the set of nodes participating in the distributed system. Let $\mathcal{D}$ be the objects domain, $\mathcal{X}_i \subset \mathcal{D}$ the set of objects that are stored in a given node $N_i \in \mathcal{N}$, and $\mathcal{X} = \bigcup \mathcal{X}_i$ the set of objects stored in the network. As in [7], our DINN is based on a priority queue:

**Definition 1.** A priority queue (Queue) is a set of pairs (element, $\vartheta$) ordered according to key $\vartheta \in \mathbb{R}^+$. An element can be either an object or a node.

In order to avoid involving all the nodes in the DINN execution, we suppose there is the possibility to evaluate a lower bound ($\delta$) for the distances between the objects stored in a certain node and any given object in $\mathcal{D}$.

**Definition 2.** Given a node $N_i \in \mathcal{N}$ and an object $x \in \mathcal{D}$ we define $\delta : \mathcal{N} \times \mathcal{D} \rightarrow \mathbb{R}^+$ as a lower bound for the distances between $x$ and all the objects stored in $N_i$ (i.e., $\mathcal{X}_i$):

$$\delta(N_i, x) \leq \min \{d(y, x), y \in \mathcal{X}_i\}$$

Note that this lower bound could even be $0$ for every node. Thus we do not strictly require this lower bound to be evaluable, but we use it for efficiency in case it can be given. In case each node $N_i \in \mathcal{N}$ of a given distributed data structure is responsible for a portion $\mathcal{D}_i$ of the domain $\mathcal{D}$ we will say that $\delta$ is strong iff:

$$\forall N_i \in \mathcal{N}, \delta(N_i, x) = 0 \iff x \in \mathcal{D}_i$$

In defining our DINN algorithm we will use the general notion of downward closed set. We will limit this notion to set of nodes with respect to a given object (using the lower bound $\delta$ defined above).

**Definition 3.** A set of nodes $\mathcal{N}_x$ is downward closed with respect to an object $x \in \mathcal{D}$ if $\forall N_j, N_i \in \mathcal{N}:

$$N_i \in \mathcal{N}_x \land \delta(N_i, x) < \delta(N_j, x) \Rightarrow N_j \in \mathcal{N}_x$$

In words, if a set of nodes is downward closed with respect to an object $x \in \mathcal{D}$, there are no nodes out of the set, with a lower bound less to that any node in the set. Another special set of nodes we will refer in the algorithm definition is the set of nodes whose lower bound $\delta$ is less than a given $r \in \mathbb{R}^+$.

Notation 1. Let $x$ be an object in $\mathcal{D}$ and $r \in \mathbb{R}^+$. $\mathcal{N}_{x,r}$ is the set of nodes in $\mathcal{N}$ that could have objects closer to $x$ than $r$, i.e.,

$$\mathcal{N}_{x,r} = \{N_i : N_i \in \mathcal{N} \land \delta(N_i, x) \leq r\}.$$

4.2 Assumptions

Our DINN algorithm is based on two assumptions.

**Assumption 1.** Eachnode $N_i \in \mathcal{N}$ is able to perform a local-INN algorithm over the objects $X_i \subseteq \mathcal{D}$, it stores.

**Assumption 2.** Let $x \in \mathcal{D}$ be an object in the domain. Let $\mathcal{N}_x \subseteq \mathcal{N}$ be a subset of the nodes that is either downward closed (with respect to $x$) or empty. Let $N_x \in (\mathcal{N} \setminus \mathcal{N}_x)$ be the closest node to $x$ in $(\mathcal{N} \setminus \mathcal{N}_x)$, i.e.,

$$N_x = \arg \min_{N_i} \{\delta(N_i, x), N_i \in (\mathcal{N} \setminus \mathcal{N}_x)\}.$$

Whenever an arbitrary node $N_i \in \mathcal{N}$ knows $N_x$, $N_x$ must be able to check if $N_x$ exists and to contact it.

Assumption 1 is needed because our DINN algorithm is built over nodes which are able to perform a local-INN.

Assumption 2 is necessary for engaging the nodes in the DINN algorithm execution as it progresses. Basically, given the lower bound $\delta$ defined in Definition 2, we require a mechanism for adding the nodes to Queue in order of decreasing distance from $q$. If, for a specific data structure, it is not possible to evaluate the lower bound $\delta$, we can consider $\delta(N_i, q) = 0$ for every node $N_i \in \mathcal{N}$. In this case the order in which the nodes are added to Queue is undefined. However in this case, we will involve all the nodes in (almost) every execution of the DINN algorithm. In fact, given that there is not a lower bound for the distance between the objects stored in a given node and the query, we can not exclude any node a priori.

The role of the subset $\mathcal{N}_x$ will be clarified in the next section which will extensively discuss the algorithm. However, we can anticipate that it represents a subset of the nodes that, at any given time during the algorithm execution, have been already asked for a local-INN execution. In particular, if $\mathcal{N}_x = \emptyset$, Assumption 2 means that any node $N_x \in \mathcal{N}$ must be able to find (using some routing mechanism provided by the distributed system), a node $N_x \in \mathcal{N}$ for which the distance $\delta(N_x, x)$ is minimum. Note that, in the discussion we never supposed that in the distributed system there is a global knowledge, but we always assume that there is a way (usually a routing mechanism) to find the most promising node for the algorithm progress. Note also that, if $\delta$ is strong, the first node added to Queue is the node $N_x$ that would contain $x$ (i.e., $\delta(N_x, x) = 0$). Therefore, in this case, the problem is similar to the lookup problem in DHTs. In case there is some replication in the distributed system, there could be two or more nodes $N_i \in \mathcal{N}$ for which $\delta(N_i, x) = 0$ is minimum. However, in this case we only need to find one of them. When $\mathcal{N}_x \neq \emptyset$, Assumption 2 means that the distributed system must be able to search for the next most promising node ($N_x$) given that we already know a set of nodes ($\mathcal{N}_x$) which are more, or equal, promising than the next one (i.e., $\mathcal{N}_x$ is downward closed).

In Section B of the Appendix, we illustrate two sufficient conditions for Assumption 2. Condition 1 can guarantee that next most promising node is always in Queue just collecting information about neighbors of yet involved nodes.
is a simpler condition easily satisfied by the proposed algorithm. In Section 4.5 we will present a message reduction optimization in case we want to retrieve next \( k^+ > 1 \) objects with a single invocation of the DINN, while in Subsection 4.6 the proposed algorithm will be extended to parallelize the operations made by distinct nodes. All the Theorems and Corollary mentioned are reported in Appendix.

To perform the DINN we need to define a node that takes the role of coordinating node (\( N_c \)). A good candidate for this role is the initiating node. If \( \delta \) is strong (see Definition 2) another candidate is the node that would store the query (i.e., \( \delta(N_c, x) = 0 \)). However, the definition of our DINN algorithm is independent on the particular choice of the coordinating node. This choice only affects the number of messages exchanged during the query execution.

As in [7] we need a Queue (see Definition 1) in which elements are ordered according to their key (see Definition 4). In Queue nodes will be assigned a different key depending whether they have already returned objects or not. Thus, we will use the following notation:

\[ \text{Notation 2. } N^* \subseteq N \text{ is the set of nodes that already performed a local-INN.} \]

An important part of the DINN algorithm definition is the definition of the keys used to order elements in Queue.

Definition 4. Given a query object \( q \in D \) we define the key \( \vartheta \) as:

\[ \vartheta_x = d(x, q), \text{ for any object } x \in D; \]

\[ \vartheta_{N_i} = \delta(N_i, q), \text{ for any node } N_i \text{ that has not yet been asked for a local-INN (i.e., } N_i \notin N^*); \]

\[ \vartheta_{N_i} = d(l_i, q), \text{ for any } N_i \in N^*, \text{ where } l_i \in X_i \text{ is the last object that } N_i \text{ returned performing its local-INN.} \]

Note that both keys used for nodes are lower bounds for the distance between the query \( q \) and the next result coming from the local-INN invocation on node \( N_i \).

The DINN algorithm consists of a loop in which:

1. If Queue is empty, the closest node (\( N_n \)) to \( q \) that has not yet performed a local-INN is added to Queue. In case \( N_n \) does not exist, the DINN is ended (there are no more objects in the distributed data structures);

2. Else, if the first element in Queue is a node (\( N_i \)), this node is asked to perform a local-INN. Then the returned result \( l_i \in X_i \) is added to Queue and the key of \( N_i \) is updated with \( \vartheta_{N_i} = d(l_i, q) \). In case \( N_i \) did not return any object (i.e., it has already returned all its objects), the \( N_i \) is removed from Queue;

3. Else, if the first element in Queue is an object \( x \), let \( N_n \) be the closest node to \( q \) that has not yet performed a local-INN and has \( \delta(N_n, q) < d(x, q) \); if \( N_n \) exists, add it to Queue, otherwise the loop is stopped returning \( x \) as next result. Note that if \( N^* \) is downward closed \( N_n \) can be found because of Assumption 2. We prove \( N^* \) to be downward closed in Corollary 1.

Queue must be kept alive for future request of more results. Obviously, the requester can close the session asserting that no more results will be asked. In this case Queue can be discarded.

In Algorithm 1 we give a DINN algorithm definition using a pseudo language. The function used in Algorithm 1 are defined as follows:

- \text{First}(Queue): returns the first element in Queue.
- \text{LocalINN}(q, N_i): asks node \( N_i \) to return the next result according to its local-INN with respect to the query \( q \).
- \text{Enqueue}(Queue, (e, \vartheta)) \): adds element \( e \), either an object or a node, to Queue with key \( \vartheta \).
- \text{Updatekey}(Queue, (N_i, r)) \): updates the key of node \( N_i \) in Queue with the value \( r \in \mathbb{R}^+ \).
- \text{Exqueue}(Queue, e) \): removes element \( e \) and its key from Queue.
- \text{Getnextnodeinh}(q, N^*, r): returns \( \arg\min_{N_i}(\delta(N_i, q), N_i \in (N^* \setminus N^*)) \).
- \text{Getnextnode}(q, N^*): returns \( \arg\min_{N_i}(\delta(N_i, q), N_i \in (N \setminus N^*)) \).

**Algorithm 1 Distributed Incremental Nearest Neighbor Algorithm**

\[
\text{loop}
\]

\[
\text{if Queue is empty then}
\]

\[
N_i \Leftarrow \text{Getnextnode}(q, N^*)
\]

\[
\text{if } N_i = \text{NULL then}
\]

\[
\text{Return NULL}
\]

\[
\text{end if}
\]

\[
\text{end if}
\]

\[
\text{else if First(Queue) is an object then}
\]

\[
x \Leftarrow \text{First(Queue)}
\]

\[
N_i \Leftarrow \text{Getnextnodeinh}(q, \langle N^*, d(x, q) \rangle)
\]

\[
\text{if } N_i = \text{NULL then}
\]

\[
\text{Return } x
\]

\[
\text{end if}
\]

\[
\text{end if}
\]

\[
\text{else if First(Queue) is a node then}
\]

\[
N_i \Leftarrow \text{First(Queue)}
\]

\[
x \Leftarrow \text{LocalINN}(q, N_i)
\]

\[
N^* \leftarrow N^* \cup N_i
\]

\[
\text{if } x \neq \text{NULL then}
\]

\[
\text{Enqueue(Queue, } (x, d(x, q)) \rangle)
\]

\[
\text{Updatekey}((N_i, d(x, q)))
\]

\[
\text{else \{node } N_i \text{ has no more objects\}}
\]

\[
\text{Exqueue(Queue, } N_i \rangle)
\]

\[
\text{end if}
\]

\[
\text{end if}
\]

\[
\text{end loop}
\]
Note that if $N^*$ is always downward closed with respect to $q$, because of Assumption 2 it is possible to implement the function GETNEXTNODE($q, N^*$). We prove this in Corollary 1. Please note also that GETNEXTNODEINR($q, N^*, r$) can be implemented using GETNEXTNODE($q, N^*$). On the other side, using GETNEXTNODEINR, we can realize GETNEXTNODE increasing $r$ until a node is found. However GETNEXTNODEINR($q, N^*, r$) can be more efficiently implemented considering that it does not need to return a node if it is farther away than $r$ from $q$.

In Appendix, we prove DINN to be optimum in terms of both local-INN invocations in Theorem 3 and number of involved nodes in Theorem 2.

4.4 Algorithm considerations

The major differences between our DINN algorithm and the INN defined in [7] are:

- Once a node comes at the head of the queue we don’t ask it to return all its objects ordered according to their distances from the query. This would be the natural extension for the INN algorithm, but, in a distributed environment, such an algorithm could not be scalable. Therefore, we ask it to return its next object using its local-INN;
- Whenever a node returns an object, we move it back in the queue using $d(l_i, q)$ as new key ($l_i$ is the last object the $N_i$ returned as a result). Please note that $d(l_i, q)$ is a lower bound for the distance between $q$ and the next result coming from the local-INN of $N_i$;
- The original INN algorithm requests a consistency condition (Definition 1 of [7]) to ensure that once a node reaches the head of the queue no other nodes can return objects with a distance smaller than the head node key. This condition has been defined for hierarchical data structure thus limiting the use of their INN algorithm. In our DINN we replaced the consistency condition with Assumption 2.

4.5 Using DINN to find next $k^+$ results

In this section we give an extension of our DINN to reduce the number of messages when we want to retrieve next $k^+ \geq 1$ objects. The price to be paid for the messages reduction is the possibility to ask a node to retrieve more objects than what is strictly necessary. At any given time during the execution of the DINN:

Notation 3. let $\hat{k}$ be the number of objects already retrieved by the previous invocations of the DINN.

Notation 4. let $k^+$ be the number of more objects we want to retrieve, and

Notation 5. let $k_{\text{ans}} \leq k^+$ be the number of results already found by the DINN during the current invocation.

If a node $N_i$ is first in Queue we ask this node to retrieve next $\hat{k}$ results where:

$$\hat{k} = k^+ - k_{\text{ans}}$$

Because $\hat{k}$ represents the number of objects we need to end the given task (i.e., retrieving next $k^+$ objects) we are sure that we will never involve $N_i$ again before the current task will be completed. Note that, by definition, $\hat{k} \geq 1$ always holds until the current task is completed.

Furthermore, we can reduce the number of unnecessary objects retrieved by the nodes considering the distance of the $k$-th object, if it exists, in Queue.

Definition 5. At any given time, let $x_1 \in X$ be the $k$-th object, if it exists, in Queue to guarantee that node $N_i$ will be involved only once during the current task, we ask node $N_i$ to perform a sequence of local-INN invocations until at least one of the following conditions is true:

- $\hat{k}$ more objects have been retrieved ($\hat{k} = k^+ - k_{\text{ans}}$);
- $d(l_i, q) \geq d(x_1, q)$, where $l_i$ is the last object retrieved;
- all the objects stored in $N_i$ have been retrieved.

The results coming from $N_i$ are added to Queue. If all the objects stored in $N_i$ have been retrieved $N_i$ is removed from Queue, otherwise its key is update with $\vartheta_{N_i} = d(l_i, q)$ and now $\vartheta_{N_i} \geq d(x_1, q)$. Either the $\hat{k}$ objects retrieved are before $N_i$ or $N_i$ is now after $x_1$. In both cases at least $k$ objects are before $N_i$ in Queue. Thus, we will not involve $N_i$ again in retrieving next $\hat{k}$ results.

In Figure 1 we give an example of Queue at a given time during the DINN execution. The dotted lines show from which node every object comes from. Suppose that we are searching for the next $k^+ = 5$ objects and we have already found next $k_{\text{ans}} = 2$ results (they are no more in Queue). We still have to search for next $\hat{k} = k^+ - k_{\text{ans}} = 5 - 2 = 3$ results. The $\hat{k}$-th object $x_3$ in Queue is $z$. Using the proposed extension, the DINN will ask node $N_3$ to retrieve objects, using its local-INN, until 3 objects have been found or the last object $l_3$, retrieved by $N_3$, has distance $d(l_3, q) \geq d(z, q)$.

4.6 Parallelization

The DINN algorithm presented in Section 4.3 always involves only the most promising node (the first in Queue). In this section we give a parallelized version of our DINN.

Generally speaking, the $k$-NN operation, is not a easily parallelizable operation as the RangeQuery is. To execute a RangeQuery, every single node can perform the search between his objects without considering the results coming from other nodes. Given the query and the range, each node can search between his objects regardless the results found in other peers. To parallelize the DINN algorithm we must accept the possibility to ask a node to give its next result even if it could be not necessary. Furthermore, in a parallelized DINN it is possible to involve nodes which would not be involved by the serial execution.

Let us assume that at a given time during the algorithm execution $x_1$ is the first object, if it exists, in Queue. In principle it is possible that we will ask all the nodes before $x_1$ in Queue to invoke their local-INN. This is true, e.g., if all these nodes return results further away from $q$ than $x_1$. To parallelize the DINN execution, we can decide to ask all the nodes before $x_1$ to retrieve the next object. We now give a definition of DINN parallelization which can be used in combination with the message optimization given in Definition 5.

Definition 6. Let $x_3 \in X$ be the $k$-th object, if it exists, in Queue and $d(x_3, q)$ its distance from the query. Given
\[ p \in [0,1] \] we parallelize the DINN asking all the nodes \( N_i \in \text{Queue} \) whose \( \vartheta_{N_i} \leq p \cdot d(x_i,q) \). In other words, using Definition 4, a node \( N_i \in \text{Queue} \) is involved if:

- \( \vartheta_{N_i} = \delta(N_i,q) \leq p \cdot d(x_i,q) \), in case \( N_i \in N \setminus N^\circ \) (i.e., \( N_i \) has not yet been asked for a local-INN);
- \( \vartheta_{N_i} = d(l_i,q) \leq p \cdot d(x_i,q) \), otherwise (i.e., \( N_i \in N^\circ \)) where \( l_i \in N_i \) is the last object that \( N_i \) returned invoking its local-INN).

Any involved node is asked to retrieve its next object invoking its local-INN. However, using the DINN optimization for k-INN search (see Definition 5), any node can be asked to perform more than one local-INN with a single message. However, in this case, there are nodes that are not at the top of Queue asked to retrieve objects. We can then consider the case in which there are objects before them in Queue. Let \( k_{N_i} \) be the objects in Queue before node \( N_i \).

In Figure 1, we give a snapshot of the priority queue at a given time during the DINN execution. As said before, the dotted lines show from which node each object comes from. As before, let us suppose that we are searching for the next \( k^+ = 5 \) objects and we have already found next \( k_{\text{total}} = 2 \) results. We still have to search for next \( k = 3 \) results. Using the proposed extension, the DINN will ask node \( N_3 \), \( N_2 \) and \( N_5 \) to invoke their local-INN and they all will work in parallel. If we also use the message reduction optimization, \( N_3 \) will be asked to retrieve at most 3 objects, while \( N_2 \) and \( N_5 \) will be asked to retrieve at most 2 objects. All of them will stop the iteration of their local-INN if \( d(l,q) \geq d(z,q) \), where \( l \) is the last object they retrieved.

Unfortunately, there could be one or more nodes \( N_i \) for which \( \delta(N_i,q) \) which are not yet in Queue. In fact, the DINN algorithm does guarantee only that the next most \( k^+ \) objects to a free node that takes responsibility for a part of the objects stored in \( N_i \) have been retrieved.

Note that, since \( k \leq k^+ \), the degree of parallelization does depend on \( k^+ \). In other words, the more objects we request at each invocation of the DINN algorithm, the greater degree of parallelization we obtain with the same \( p \).

In the case \( x_i \) does not exist (i.e., there are less than \( k \) objects in Queue), we invoke just the first node (which is at the top of Queue). Once \( x_i \) does exist in Queue, the parallelization is used again.

Another choice, in case \( x_i \) does not exist, is to use the distance of the last object in Queue in place of \( d(x_i,q) \). In this case the operation would become more parallel but also more expensive considering its total cost. The degree of parallelization of the DINN is also related to the number of objects present in Queue. Thus, it is important to have more than \( k \) objects in Queue. Different strategies can be used to efficiently put nodes in Queue depending on the specific data structure that is used.

5. DINN OVER MCAN

The MCAN [5, 6] is a scalable distributed similarity search structure for metric data. MCAN is able to perform distributed similarity searches between objects assuming that the objects, together with the used distance, are metric. For a complete description of MCAN see [5]. A comparison of MCAN with similar distributed similarity search structure for metric data can be found in [3].

MCAN satisfies Condition 1 which guarantees Assumption 2 as demonstrated in Theorem 2. In fact, it can be proved that in a MCAN, if a node \( N_i \) is neighbor of a node \( N_j \) that is closer to the query than \( N_i \) and \( \delta(N_j,q) > 0 \), then \( N_j \) is also neighbor of at least one other node which is closer to the query than \( N_j \). In other word MCAN satisfies Assumption 2. In fact, given a set of nodes \( N^\circ \) downward closed with respect to \( q \), the node \( N_0 \) (see in Condition 1) is always between the neighbors of at least a node \( N_i \in N^\circ \).

5.1 Experimental Results

Experiments have been conducted using a real-life dataset of 1,000,000 objects. Each object is a 45-dimensional vector of extracted color image features. The similarity of the vectors was measured by a quadratic-form distance [10]. The same dataset have been used for, e.g., [6, 3, 9, 1]. The dimensionality used for the MCAN is 3 as in [6]. All the presented performance characteristics of query processing have been taken as an average over 100 queries with randomly chosen query objects.

To study scalability with respect to the number of objects, we limited the number of objects each node can maintain (the same has been done in [2, 5, 6, 3, 9]). When a node exceeds its space limit it splits by sending a subset of its objects to a free node that takes responsibility for a part of
we show the estimated cost for retrieving up to 500 objects 10 by 10 (i.e., \(k^+ = 10\)) comparing the defined \(DINN\) with a stateless execution of the \(DINN\) in which after searching first 10 objects we destroy \(Queue\) and then we ask for next 10 objects (thus requesting a 20-NearestNeighbor search from scratch) and so on. What we want to underline is that the use of a Incremental Nearest Neighbor when the number of desired neighbors is unknown in advance is mandatory to preserve efficiency. In fact the cost of retrieving...
we report the costs for growing dataset. Let’s now consider the parallelized version of the DINN defined in Subsection 4.6. In Figure 7 we compare the total and parallel cost when \( p = 1.0 \) (i.e., maximizing the parallelization). The graph of the parallel cost demonstrates the advantage of the parallel execution. Observing for instance \( k = 100 \) for the case \( k^+ = 10 \), the parallel cost is slightly larger than 100, while for the same case the sequential cost (Figure 5) is about 1300. \( k^+ = 10 \) seems a good trade off between the total and the parallel cost. In fact, the total cost is almost the same as of the sequential case.

Another set of experiments were conducted by varying \( p \) from 0 to 1 for a growing dataset. In this experiments we fixed \( k = 500 \) and used various \( k^+ \).

In Figure 8(a) we report the costs for growing dataset, number of results \( k = 500 \) and \( k^+ = 1.0 \). The total cost does not significantly vary with \( p \), i.e., parallelization, for \( k^+ = 1 \), is obtained without increasing the total cost. Another important aspect is that parallel cost is slightly influenced by the dataset size when the parallelization degree is maximum \( (p = 1) \).

In Figure 8(b) we report the costs for growing dataset, \( k = 500 \) and \( k^+ = 10 \). We can see that increasing \( k^+ \) the differences between the parallel costs of different degree of parallelism \( (p) \) are more relevant. However, the total cost for different \( p \) are very similar and almost the same of the ones obtained for \( k^+ = 1 \) in Figure 8(a). It is also important to observe that the parallel cost does scale, with respect to the dataset size, for \( p = 1 \) and just a little bit less for \( p = 0.5 \).

Finally, in Figure 8(c) we report the costs for \( k = 500 \) and \( k^+ = 50 \). In this case the parallel cost is better than for the \( k^+ = 1 \) case but the total cost does depend on \( p \). However, the most important result is that the parallel cost not only scale with respect to the dataset size, but it slightly decreases. Obviously, this is possible because we are adding more resources (nodes) as the dataset size increase (proportionally), but this should be common in a P2P environment where typically more nodes means more data and vice-versa.

In Figure 9 we report the percentage of involved nodes for \( k^+ = 10 \) as the dataset grows. As expected, the more parallelism, the greater percentage of involved nodes. However, it is interesting to notice that results for \( p = 0.5 \) and \( p = 1.0 \) are almost the same. Considering scalability with respect to the dataset size, it is important that the percentage of involved nodes does decrease with the number of objects, i.e., with the number of nodes.

6. CONCLUSIONS AND FUTURE WORK

Distributed incremental nearest neighbor search is a big challenge for at least two reasons. It is quite handy to have a possibility to easily increment the number of nearest neighbors at a low cost instead of being forced to an expensive solution of specifying high values of \( k \) to ensure having enough objects in all situations or staring the nearest neighbor search over and over again whenever the value of \( k \) grows. Second, distributed environments do not allow application of existing centralized solutions and completely new solutions are needed.

In this paper, we have defined a distributed incremental nearest neighbor especially suitable for structured P2P similarity search networks. The proposed algorithms have been implemented in a large network of computers using MCAN and extensively tested on a real-life data collection: color features of images. We proved our algorithm to be optimum in terms of both the number of involved nodes and the number of local-INN invocations when executed in a serial way. However, our algorithm also allows controlling the degree of parallelism by using a special parameter.

As a next step of our research, we plan to apply this distributed incremental nearest neighbor search to other distributed similarity search structures, such as GHT*, VPT*, or M-Chord. Naturally, this incremental approach will vitally be important in developing multi feature similarity search execution strategies, such as we need for the top \( k \) multi-feature queries.
7. ACKNOWLEDGMENTS

This work was partially supported by the SAIPR (Search In Audio Visual Content Using Peer-to-Peer IR) project, funded by the European Commission under IST FP6 (Sixth Framework Programme, Contract no. 45128).

8. REFERENCES


APPENDIX

A. CORRECTNESS AND OPTIMALITY

Corollary 1. At any time during the DWDN algorithm execution, the set of nodes \( N^{*} \) (i.e., the set of nodes that already performed a local-INN) is downward closed with respect to the query \( q \).

Proof. We prove the corollary using induction. When the algorithm starts Queue is empty and a node \( N_i \) is added to Queue using Getnextnode(q, \( \emptyset \)) (usually \( \delta(N_i, q) = 0 \)). After \( N_i \) has been asked for a result, \( N^{*} \) contains only \( N_i \) and is downward closed by definition of Getnextnode. At a given time in the algorithm execution, let \( N_i \) be the node, if it exists, returned by the function Getnextnodeinr(q, r, \( N^{*} \)) or by the function Getnextnode(q, \( N^{*} \)). Because of the functions definitions, if \( N_i \) exists, there is no other node \( N_j \in N^{*} \) for which \( \delta(N,j, q) < \delta(N_i, q) \). Then \( (N^{*} \cup N_i) \) is still downward closed with respect to \( q \).

Theorem 1 (Correctness). Let \( R \) be the set of objects already returned by the DWDN algorithm. Whenever DWDN returns an object \( x \) there are no objects nearer to the query:

\[
d \in R \quad \forall y \in X \quad d(y, q) < d(x, q) \Rightarrow y \in R
\]

Proof. By definition of \( X \) there must be a node \( N_i \in N^{*} \) for which \( x \in X_i \) . Using Notation 1, \( d(y, q) < d(x, q) \Rightarrow N_j \in N_{x,y} \). Because of the algorithm definition, Getnextnodeinr(x, d(x, q), \( N^{*} \)) did not return any node. Then, by Getnextnodeinr definition, \( \{N_{y,d(x,q)} \setminus N^{*}\} = \emptyset \) and then \( N_j \in N^{*} \) (i.e., \( y \) belongs to a node which has already been asked for a local-INN). If \( N_y \in N^{*} \) has some not returned objects by algorithm definition \( N_i \) is in Queue with key \( d(l_1, q) \) (where \( l_1 \in X_i \) is the last object it returned). Because \( x \) is first, \( d(l_1, q) > d(x, q) > d(y, q) \). Then \( y \) must be between the objects \( N_i \) already returned, which are either in \( R \) or in Queue. But \( y \) cannot be in the priority because \( x \) is first and objects are ordered according to their distance, then \( y \in R \).

Theorem 2. The DWDN is optimal with respect to number of involved nodes given the lower bound \( \delta \) to exclude nodes.

Proof. The algorithm can be rewritten as follows. Let \( N^{*} \) be the set of involved nodes, \( x \in X \) the last object returned by the DWDN and \( q \in \mathbb{D} \) the query object. If the local-INN is invoked in \( N_i \), the lower bound of the distance between \( q \) and the objects in \( N_i \) is less than the distance between \( q \) and \( x \), i.e.,

\[
N_i \in N^{*} \Rightarrow \delta(N_i, q) \leq d(x, q).
\]

Because of the algorithm definition (see Algorithm 1), \( x \) was the first element in Queue and each node is requested to perform a local-INN result only when they are at the head of Queue. Because of \( \delta(N_i, q) \) and \( d(x, q) \) are used as key for not yet involved nodes and objects respectively (see Definition 4), the last equation always olds.

Theorem 3. The DWDN is optimal with respect to the number of local-INN invocations given the lower bound \( \delta \) to exclude nodes and using the lower bound \( d(l_1, q) \) for the distance between the query \( q \) and the next result coming from the \( N_i \) local-INN.

Proof. In Theorem 2 we proved that the DWDN is optimum in terms of number of involved nodes. Thus, DWDN is optimum in terms of local-INN first invocations. Moreover, being \( d(l_1, q) \) the key (used to order elements in Queue) for a node \( N_i \) that already performed a local-INN (see Definition 4), whenever \( N_i \) is asked to retrieve its next result (using its local-INN) we are sure that the DWDN next result will be further away than \( d(l_1, q) \). In fact, we are using as key in Queue (see Definition 4): \( d(x, q) \) for every object \( x \); a lower bound for \( d(y, q) \) for every node \( N_i \).

B. SUFFICIENT CONDITIONS FOR ASSUMPTIONS 2

Condition 1. Let \( N_q \) be a downward closed set respect to an object \( q \in \mathbb{D} \). For any given \( N_i \in \mathbb{N} \), let \( N_i \subseteq N^{*} \) be the set of nodes which \( N_i \) is able to contact directly independently from the execution of the current DWDN algorithm. Let \( N_q \subseteq \mathbb{N} \) be the closest node to the query (according to \( \delta \) which is not in \( N_q \) (as defined in Assumption 2). If \( N_q \) exists, it is in the union of the set of nodes known by the nodes in \( N_q \):

\[
N_q = \arg \min_{N_q} \{\delta(N_i, q), N_i \in (N_q_r \setminus N_q) \} \in \bigcup(N_i, N_i \in N_q).
\]

Proof. By Condition 1, \( N_q \) can ask each node \( N_i \in N^{*} \) which are its neighbors \( (N_i) \). Sorting the union of them \( (\bigcup(N_i, N_i \in N_q)) \) \( N_q \) is able to find \( N_q \).

Condition 1 basically says that it is always possible to pass from one node \( N_{q-1} \) to the next one \( (N_q) \) just using the information we found in the previous nodes. The information we need is the knowledge they have about other nodes (typically neighbors). This condition is very useful to efficiently implement Getnextnode.

Condition 2. For any given object \( q \in \mathbb{D} \) and \( r \in \mathbb{R}^{+} \), every node \( N_i \in \mathbb{N} \) is able to know all the nodes (their addresses) in \( N_{q,r} \).

Proof. By Condition 2, \( N_c \) can ask for all the nodes in \( N_{q,r} \). If \( (N_{q,r} \setminus N_q) = \emptyset \), the next node \( N_q \) is the nearest to the query in \( (N_{q,r} \setminus N_q) \). Otherwise, if \( (N_{q,r} \setminus N_q) \neq \emptyset \), \( N_q \) can try again increasing \( r \) until \( r \leq d_{\text{max}} \). In this last case \( N_q \) does not exist.

Please note that all distributed data structures able to perform a range query, should be able to satisfy Condition 2 and theorem Assumption 2). Under Condition 2 Getnextnodeinr is efficiently implemented while Getnextnode can be realized increasing \( r \) until either a node is found, using Getnextnodeinr; or \( r \) exceeds the max possible value of \( d \) (i.e., \( d_{\text{max}} = \max(d(y, x), y, x \in D) \)).