On Efficiently Processing Nearest Neighbor Queries in a Loosely Coupled Set of Data Sources

Thomas Schwarz, Markus Iofcea, Matthias Grossmann,
Nicola Hönle, Daniela Nicklas, Bernhard Mitschang

University of Stuttgart, Institute of Parallel and Distributed Systems (IPVS)
Universitätsstrasse 38, D-70569 Stuttgart, Germany
+49-711-7816-411

thomas.schwarz@informatik.uni-stuttgart.de

ABSTRACT

We propose a family of algorithms for processing nearest neighbor (NN) queries in an integration middleware that provides federated access to numerous loosely coupled, autonomous data sources connected through the internet. Previous approaches for parallel and distributed NN queries considered all data sources as relevant, or determined the relevant ones in a single step by exploiting additional knowledge on object counts per data source. We propose a different approach that does not require such detailed statistics about the distribution of the data. It iteratively enlarges and shrinks the set of relevant data sources. Our experiments show that this yields considerable performance benefits with regard to both response time and effort. Additionally, we propose to use only moderate parallelism instead of querying all relevant data sources at the same time. This allows us to trade a slightly increased response time for a lot less effort, hence maximizing the cost profit ratio, as we show in our experiments. Thus, the proposed algorithms clearly extend the set of NN algorithms known so far.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Distributed databases; Query processing*; H.2.8 [**Database Management**]: Database applications—*Spatial databases and GIS*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process.*

General Terms

Algorithms, Performance, Experimentation.

Keywords

Data integration, distributed query processing, federated database system, kNN, nearest neighbors, parallel query processing.

1. INTRODUCTION

Various applications use spatial data: location-based services show points-of-interest on a map, spatial data mining applications look for the best place to establish a new shop, and so forth. In most of these scenarios the spatial data is investigated using two types of queries: window queries and nearest neighbor (NN) queries.

Window queries take a spatial predicate (like *intersects* or *inside*) and a query region (a *rectangle* or a *polygon*) as input parameters. All objects qualify that satisfy the spatial predicate comparing the query region to the object's geometry (e.g., the objects of a map region).

NN queries take as input parameters a reference point and the number of results to return (commonly referred to as *k*). Only the *k* objects closest to the reference point are returned (e.g., the five nearest gas stations).

1.1 Motivation

So far NN queries have been addressed to a single, in most cases local, data set. In this paper we focus on querying a loosely coupled set of *autonomous* spatial data sources where we have *no influence* on the distribution of the data across the sources. The result of a federated nearest neighbor query (FNNQ) has to contain the *k* nearest objects from all available data sources. The algorithm introduced in this paper uses a minimized number of remote queries involving only a fraction of the data sources to process a FNNQ and does not rely on exact object counts and density statistics. This is an important aspect, because it is difficult and expensive to keep such statistics up-to-date due to the autonomy of the data sources.

Let's take a look at a location-based services scenario to highlight the benefits of an integrated NN query facility. Numerous servers running spatial databases store data on restaurants: each fast food restaurant chain has its own server, local restaurants transfer their data to a web-hoster setting up a logical server for each restaurant, a city's yellow pages server features restaurants, and local magazines offer restaurant reviews on their servers. A hungry user does not want to choose a particular server, or compare results from different servers, but he wants to ask them all at once and receive an integrated result.

Our algorithm uses a spatial directory, managing the list of all available spatial data sources, to determine the relevant ones for each query. The directory stores the name of a data source, its connection parameters and its service area, which is a region containing all spatial objects of the data source. As an implementation of such a directory an OGC Catalog Service

[19, 20], the FGDC clearinghouse [11], or even a spatially enabled web services directory service [21, 26, 28] can be used.

Our algorithm is targeted at data integration scenarios, where the data cannot be replicated locally due to at least one of the following three reasons:

- The size of the accumulated data is too large.
- All data sources are autonomous and each one disallows to replicate its contents, e.g., because of access restrictions.
- Propagating updates to local replicas is too expensive.

In any of the above cases the preferable solution is to employ an integration middleware that federates the participating data sources.

1.2 Objectives and Contributions

In this paper we introduce new solutions to the problem of processing NN queries in a federation of many loosely coupled, remote spatial data sources, and we present and analyze a family of algorithms in detail. The data is tied to the sources and we cannot change its distribution. We compare different granularities of statistics available to the integration middleware: none, rough global density estimates, and detailed object counts per data source. The data sources are accessed using either remote NN queries or remote window queries. We propose to use the degree of parallelism as a controlling element to achieve either minimal response time or minimal resource consumption. We define four performance metrics to assess the efficiency of the algorithm: response time, effort, iterations, and cost profit ratio. We ran experiments with different characteristics on random data sets to show the scalability of our approach. We can show that the cost profit ratio improves up to 3.5 times over the best approach from literature.

The remainder of this paper is organized as follows. In Section 2 we describe the system environment of our algorithm in detail and relate it to previous work on NN queries. We present the core algorithm in Section 3 along with the helper functions to calculate query range estimates and to query remote servers. In Section 4 we describe the settings and the parameters of the experiments and detail on the experiments' results. The paper concludes with directions of future work.

2. PROBLEM SPACE

The problem space of *k* nearest neighbor (NN) queries[1] can be divided into four different categories. We first describe the category of federated nearest neighbor queries (FNNQs) and our data integration scenario in the next section. Thereafter, we discuss the other categories (local NN queries, remote NN queries, and distributed and parallel NN queries on declustered data), and comment on their significance to our FNNQ solution. We focus on 2D point objects, but our algorithms can be easily generalized to process objects having an extent in higher dimensional space.

[1] We focus on the general case to find *k* nearest neighbors and neglect the special case where *k = 1*.

2.1 Federated Nearest Neighbor Queries (FNNQ)

FNNQs look for the *k* nearest neighbors to a given reference point within the data of many loosely coupled, remote spatial data sources containing overlapping and complementary data. Figure 1 shows several data sources containing similar objects (e.g. information on restaurants).

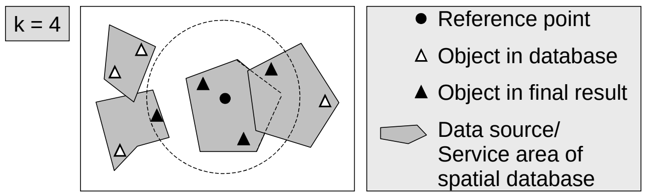

Figure 1. Restaurant objects distributed across several data sources (service areas may overlap, each object belongs to the data source inside whose service area it is located)

In this data integration scenario the user wants to find the closest restaurants available in any data source. The algorithm presented in Section 3 allows to maintain the facade of a single data source while leaving the data at its origin.

As shown in Figure 2, a system architecture for processing FNNQs consists of three different types of components [17]: spatial data sources, a spatial directory, and a federation component.

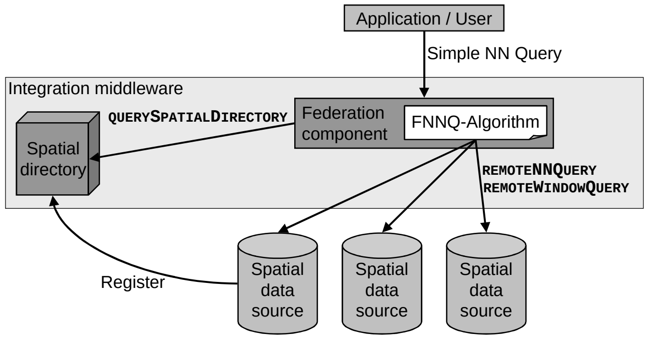

Figure 2. System architecture

The data sources offer at least one of the following two interfaces: a remote window query or a remote NN query interface. They are registered at the spatial directory with their service area[2] and the connection parameters necessary to access their query interfaces.

The spatial directory is used by the federation component to discover the relevant data sources for each query. For this, it stores the service areas of all registered data sources (which is comparable to storing the inner nodes of a global spatial index,

[2] The service area surrounds all objects stored in the data source. Its shape may be any Simple Features geometry [18]. Typically, it is a polygon following administrative boundaries of, e.g., a city district or a state, or it is the convex hull around the data source's contents. The sizes of two data source's service areas may differ by several orders of magnitude.

where the data sources represent the leaves). Depending on the method to determine the initial query range, it additionally stores a global object density estimate or the number of objects managed by each data source. As the service areas change only rarely, the spatial directory can be easily duplicated to achieve scalability. This is less practical if it stores the more frequently changing object counts.

The federation component runs the FNNQ algorithm. Towards the applications it presents the interface of a simple NN query [12, 16, 27]. It forwards queries to the remote spatial data sources and integrates their results. It can be easily duplicated for scalability as it does not store any data itself.

To assess the efficiency of our algorithm, we define two optimization goals: minimal response time, as perceived by the application, and minimal resource consumption, considering the effort to calculate query results at the data sources and to transmit them to the federation component. While an application strives for minimal response time, a system operator favors minimal resource consumption. We point out how to achieve each goal and how to find a good tradeoff between them.

2.2 Local Nearest Neighbor Queries

Local NN queries have direct and local access to the data and its indexes. Previous work has concentrated on developing algorithms either to efficiently traverse the nodes of an index tree [5, 12, 13, 27], like an R*-Tree [4], and to prune nodes as soon as possible, or to construct local indexes [1, 3, 6, 29]. See [16] for an extensive discussion of local approaches. They are restricted to a local processing and to a locally available index, and therefore they are not extensible to remote data sources. As mentioned in Section 1.1, a full copy of the data sources to a local node for constructing an index is not viable either. However, these approaches can be used within a data source to process a remote NN query locally.

2.3 Emulated Nearest Neighbor Queries

Emulated NN queries access a single remote spatial data source using window queries to emulate a NN query [16]. The goal is to minimize the number of window queries and the number of retrieved objects by estimating the size of the query window as precisely as possible. We employ their range estimation methods, see Section 3.4 for details.

Their approach can be easily adapted to a federated data integration environment: In each iteration the window query is first sent to the spatial directory and then to all relevant data sources. Our *RWQ_ObjDens* and *RWQ_ObjCnt* variants represent the outcome of this adaptation (see Section 3.5.2), and we compare them to the variants using remote NN queries in Section 4. In contrast to [16], which uses rectangular query windows, we allow them to be regular n-corner polygons to reduce surplus intermediate results. Additionally, we introduce pruning mechanisms to skip distant servers and to shrink the query window as soon as *k* results, but not the *k* closest ones, have been found.

2.4 Distributed and Parallel Nearest Neighbor Queries

Previous work on distributed and parallel processing of similarity queries has focussed on how to decluster the data [2, 7, 8, 9, 15, 22]: they partition the dataspace and distribute the data cleverly to different disks, so that for any given query most of the data can be fetched in parallel.

In our scenario, we have neither influence on the distribution of the data across the data sources nor on the shape of the dataspace's partition covered by a data source. Also, partitions may overlap. Only a fraction of the data sources contain relevant data for a given query, and we focus on querying as few of them as possible. In contrast to the above approaches, which access all disks at the same time, we propose to control our algorithm's parallelism by adapting the number of threads used in each iteration. This serves as a controlling element to make the algorithm run either with minimal response time, or with minimal resource consumption, or to achieve a good trade-off in between. Thus, the above approaches may be employed to process NNQs locally at a data source, but not to efficiently process NNQs in the federation component.

In [23, 25] several algorithms are proposed that aim at minimizing the number of source databases to be queried and the number of objects to be fetched from each system. They use a distributed R-Tree approach where the upper part is stored at the primary server and the leaves of the tree are stored at source databases. However, they rely on object count statistics to determine the set of relevant source databases, and they query them either one at a time or all at once. They provide no experimental results for a large-scale system (more than 10 databases) where each database is responsible only for a small portion of the dataspace. The *RNNQ_ObjCnt* variant (see Section 3.3) represents these algorithms in our experiments (see Section 4.2).

A similar primary server / source databases approach is presented in [14]. They try to minimize the response time by parallelizing the query processing as much as permitted by network transmission throughput. For this, they determine the optimal chunk size, corresponding to the size of clusters of close-by objects that get stored on the same source database. However, we have no influence on the data distribution, and we seek to avoid activating all promising servers at the same time.

In [24] the nodes of the R-Tree index structure are distributed among several disks. In their terms, our distributed R-tree has just two levels: the root node (spatial directory) and the leaf nodes (source databases). When traversing the R-Tree all relevant child nodes of the current node are accessed in parallel. Thus, their algorithm provides no benefit in our scenario.

[10] addresses a scenario where an object's data itself is partitioned and distributed to different systems. They aim at minimizing the number of candidates fetched from each system in order to find the globally most similar objects. However, this algorithm is not applicable to our scenario as we assume that all of an object's data is stored at a single system.

To put it in a nutshell, previous approaches address data declustering, rely on object counts, access all data sources at the same time, or target a different system architecture. In contrast to this, our approach has no influence on the distribution of data across servers, prefers object density statistics, utilizes a moderate degree of parallelism, and – if necessary – uses remote window queries to compensate for the lack of NN query functionality at a data source. The distinguishing features of our algorithm are summarized in Table 1.

Table 1. Distinguishing parameters of the FNNQ algorithm compared to other approaches

Parameter	Other approaches from literature	Proposed FNNQ algorithm
Data distribution	• optimal distribution calculated using declustering algorithm	• no influence on distribution (autonomous data sources)
Data access method	• all data items in the disk block associated to an index entry are read from disk • remote NN queries	• remote NN queries • remote window queries
Determination of relevant sources	• all sources are considered relevant as data is declustered to maximize parallel I/O • object counts (which are expensive to keep up-to-date in our scenario) and service areas are used to determine the minimum set of sources guaranteeing to contain all results	• just service areas (to reduce the number of sources queried) • service areas and global object density estimate (to additionally reduce the number of iterations)
Iterations	• one, all relevant sources are determined in a single step	• several, the set of relevant sources is iteratively refined
Parallelism	• maximum: all relevant sources are queried at the same time • none: sources are queried one after the other	• moderate degree of parallelism: reduces resource consumption while still minimizing the response time

Our experiments in Section 4 will show that the most influential parameters are:

- Parallelism: cost profit ratio improves up to 2.9 times.
- Determination of relevant sources: cost profit ratio improves up to 2.5 times.
- Data access method: cost profit ratio improves up to 2.0 times.

3. FEDERATED NEAREST NEIGHBOR QUERY ALGORITHM

The FNNQ algorithm takes a reference point (`RefPoint`) and the total number of results to retrieve (*k*) as input parameters. It returns those *k* objects from all connected data sources that are closest to the reference point. By adjusting the degree of parallelism, the algorithm can be adapted to achieve an optimal tradeoff between response time and resource consumption.

In our FNNQ algorithm we use the following external functions:

- Basic geometry functions (`DIST`, `BUFFER`, `CONTAINS`, `INTERSECTS`) as described in [18].
- A function to query a spatial directory which returns a list of all servers (data sources) whose service area intersects with the given query area.

3.1 The Three Phases Of The FNNQ Algorithm

Figure 3 demonstrates the operation of the core algorithm for the case that no additional statistics are available. The core algorithm consists of the following phases:

- The **initial phase** is the first iteration of the core loop (Figure 4). An initial range is estimated and all data sources with intersecting service areas to this range are queried. We discuss four ways to get an initial range estimate in Section 3.3.
  In our example shown in Figure 3 the initial range is the reference point itself. Hence, data source A is queried and it returns objects 1 and 2.

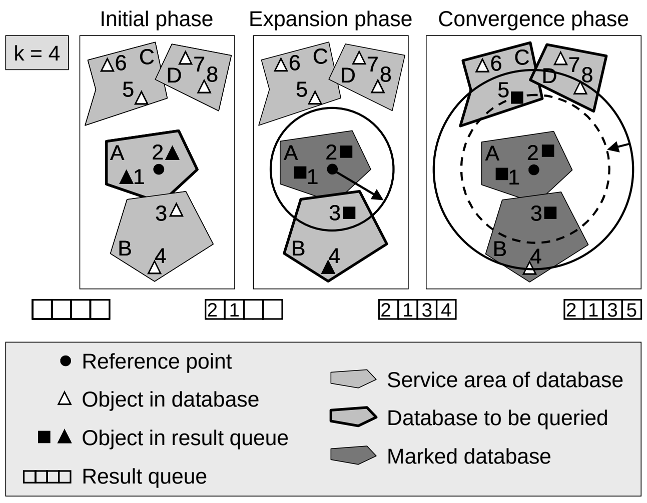

Figure 3. The three phases of the FNNQ algorithm (*k* = 4)

- In the **expansion phase** the query area is iteratively enlarged until *k* objects are found. The density of the objects fetched so far is used to determine the new query range (see Section 3.4).
  In Figure 3 the query range is enlarged. Data source B is queried and it returns objects 3 and 4.
- The **convergence phase** is required if *k* objects have been found and at least one of them is outside the query area (like object 4). It looks for even closer objects than the ones found so far. In this last iteration of the core loop the query range is set to the most distant object in the result queue and it shrinks when closer objects are found.
  In Figure 3 the query area's radius is enlarged to the distance of object 4. Data sources C and D are queried and return objects 5 and 8. Object 5 replaces object 4 in the result queue, because it is closer.

Only the initial phase is mandatory, the other two can be skipped under certain circumstances.

3.2 The FNNQ Algorithm Core Loop

Figure 4 shows a pseudo-code description of our FNNQ algorithm. We describe the functions used in the algorithm in the following sections.

```
   FEDERATEDNEARESTNEIGHBORQUERY(RefPoint, k)
   : ResultsPQ
1  IterCounter := 0
2  // don't ask these servers again:
   MarkedServersList := new List()
3  // priority queue of result objects (ordered by distance to RefPoint):
   ResultsPQ := new PriorityQueue(k)
4  // determine query area's range in initial phase:
   Range := GETINITIALRANGE(RefPoint, k)
5  do
6    IterCounter := IterCounter + 1
7    QueryArea := RefPoint.BUFFER(Range)
8    // query spatial directory for servers whose service areas intersect
     // with the query area, determine non-marked ones, and order
     // them by the distance of their service areas to the RefPoint
     ServersPQ := SETUPSERVERSPQ(QueryArea,
                    MarkedServersList, RefPoint)
9    // query these servers for objects and add objects to
     // global result queue ResultsPQ
     QUERYSERVERS(RefPoint, k, QueryArea,
        ServersPQ, ResultsPQ, MarkedServersList)
10   // termination conditions to prevent an endless loop:
     if ((MarkedServersList.size
            >= TotalNumberOfServers) or
   (QueryArea.CONTAINS(UnionedAreaOfAllServers))
        or (IterCounter >= MaxIters))
11     break
12   endif
13   // determine new range of the query area to use
     // in the next iteration:
     OldRange := Range
14   Range := GETNEXTRANGE(RefPoint, k,
                      ResultsPQ, OldRange)
15 // repeat the core loop while not enough objects are found
   // or the range is increasing:
   while ((ResultsPQ.size < k)
          or (Range > OldRange))
16 return ResultsPQ
```

Figure 4. FNNQ algorithm core loop

3.3 Determining The Initial Query Areas (GETINITIALRANGE)

The initial query area is used in the first iteration of the core loop. In our experiments we compare four different methods to determine the initial query area. Two methods get along with no additional information besides service areas:

- **Zero**. Use the reference point itself as query area in the first iteration, i.e., the initial query range is zero. Possibly no data sources are queried at all in the first iteration.
- **Maximum**. Use the infinite distance as query range. This guarantees that all relevant data sources are queried in the first iteration. Most of the approaches discussed in Section 2.4 use this method as it maximizes parallelism and promises the shortest response times.

The other two methods exploit additional metadata about the data stored at the sources:

- **Object Density**. Use a global estimate of the density of all objects in the dataspace to approximate the query range so that statistically *k* objects are enclosed in the query area. This method works best if the objects are uniformly distributed across the dataspace. It leads to additional iterations of the core loop if this assumption is not exactly satisfied. [16] refers to this method as the "Density-Based Method".
- **Object Count**. Use the exact number of objects stored on each data source to calculate the minimal query area that guarantees that all objects are found in the first iteration. For this, all data sources are ordered by the maximum distance of their service area to the reference point. Then, this list is traversed in increasing order and the data sources' object counts are added up. The traversal stops when this sum gets larger than or equal to *k*. The maximum distance of the last data source is taken as the range of the initial query area. Besides being expensive to keep the object counts up-to-date, this guaranteed single iteration behavior leads to a lot of extra work. [16] refers to this method as the "Bucket-Based Method", [24] calls this range the threshold distance, [23] uses this range in the "Parallel Nearest Neighbor Finding Method".

We propose, that in our scenario only the Zero and the Object Density methods are reasonable. The Maximum method is inappropriate with a large number of servers, and the Object Count method fails, because it is hardly conceivable that all independent data sources publish exact and current object counts. The requisites and characteristics of the four methods are summarized in Table 2.

Table 2. Methods to determine the initial query area's range

Method	Abbre-viation	Requisites	Characteristics
Zero	Zero	None	May not find any sources initially
Maximum	Max	None	Only one iteration, yields far too many relevant sources
Object Density	ObjDens	Global density estimate	Tends to underestimate the set of relevant sources
Object Count	ObjCnt	Object count per data source	Only one iteration with lots of relevant sources

3.4 Determining The Subsequent Query Areas (GETNEXTRANGE)

In the expansion phase, the density of the objects found in previous iterations is used to calculate the query area for the next iteration. Thus, the query area quickly adapts to the real distribution of the qualifying objects, even in the case of outdated statistics. If no objects have been found so far, the previous range estimate is doubled. If the previous range estimate is zero, a heuristically determined, fixed value is used.

The expansion phase ends when *k* objects are found. It is skipped if the initial phase has already yielded *k* objects. If the *k*-th object lies inside the last query area, the convergence phase is skipped. Otherwise, the query area is enlarged to include the *k*-th object and the core loop of the FNNQ algorithm is executed one more time.

3.5 Querying The Data Sources (QUERYSERVERS)

The following two alternative algorithms assume the list of servers to be queried (`ServersPQ`) to be sorted in ascending order of their distance[3] to the reference point. Note that the result objects queue (`ResultsPQ`) is limited to *k* entries. If more objects are added, the surplus objects at the end of the queue are removed.

3.5.1 Using Remote Nearest Neighbor Queries (RNNQ)

For each server in `ServersPQ`, a local refinement of the query parameter *k* is determined (`LocalK`). It indicates the maximum number of objects this server can contribute to the final result. `LocalK` is calculated by subtracting the number of those objects in the result queue, that are closer to the reference point than the server's service area, from *k*. This pruning mechanism considerably reduces the number of objects requested from a server and minimizes the number of objects discarded later in the processing. If `LocalK` is zero, a server cannot contribute any objects. The server is skipped. Otherwise, this server is queried for `LocalK` nearest objects and the new objects are added to the result queue. The server is marked as queried (added to `MarkedServersList`).

3.5.2 Using Remote Window Queries (RWQ)

If data sources do not support remote NN queries, instead, as proposed in [16], remote window queries can be used. A query for objects within the query area is sent to each server in `ServersPQ` and the resulting objects are added to the result queue. A local refinement of the query area is calculated, if `ResultsPQ` contains at least *k* objects. Its range (`LocalRange`) is equal to the distance of the *k*-th object. A server is skipped if the minimum distance of its service area is smaller than `LocalRange`. The server is marked as queried, if its service area lies completely inside the query area.

This approach has two disadvantages. First, a server may be queried multiple times until the query area is large enough to contain all its objects. Secondly, a server may return far too many objects.

3.5.3 Querying in parallel

The QUERYSERVERS function can be parallelized. We create a pool of threads, each thread running the original QUERYSERVERS function. More threads lead to a shorter total response time of the QUERYSERVERS function, but also to less efficient pruning mechanisms (`LocalK` and `LocalRange`), so that less servers are skipped and more objects are fetched. Hence, the overhead increases and more objects are needlessly retrieved. The response time is minimal if all servers are queried concurrently. On the other hand, the fewest resources are consumed if the servers are queried successively, i.e., with only one QUERYSERVERS thread. In Section 4.2 we detail on how to achieve a good trade-off.

3.5.4 Querying incrementally

Retrieving nearest neighbors incrementally from each relevant site and then performing a merge algorithm for the produced objects until we have retrieved k result objects seems to be the most straightforward algorithm to process FNNQs. However, retrieving the next object (incrementally) from a remote data source involves almost the same communication delays as issuing an entire query and retrieving all results. As illustrated in Section 4, communication delays provide a major contribution to the overall response time. Already with small k this approach involves prohibitively many interactions between the federation component and data sources, greatly increasing the response time. Hence, we refrain from using this approach in our experiments.

4. EXPERIMENTS

In our experiments we demonstrate the performance of the proposed algorithm under various conditions by varying the query parameter *k* and by varying the number of threads in the QUERYSERVERS function. The experiments are based on a simulation approach where the algorithm runs in full detail and the infrastructure is provided by local components simulating the remote ones. We had to revert to the simulation approach as we do not have enough real servers at our disposal to run internet-scale experiments. We generate a load of 1000 queries with random reference points. All diagrams show the mean values measured with these queries. The characteristics of the experiments are shown in Table 3.

Unfortunately, no freely available real world data set is both sufficiently large to run significant experiments and is divided up into overlapping partitions resembling different providers offering similar information. Also, using the US Census Bureau's Tiger/Line data is not beneficial as we still have to impose an artificial partitioning on the real data and distribute it across different providers having overlapping service areas.

Thus, the scenario's characteristics are best reflected if the simulated data sources store random data sets. Each data source gets a randomly generated polygon as its service area. The objects are uniformly distributed within the data space. For each object all data sources with suitable service areas are determined and one of them is randomly picked to host the object.

For each query its response time is calculated by adding up the per query part, accounting for typical communication delays and latency in the internet, and the per object part, accounting for the local processing effort and transportation costs. In [23] it is experimentally approved that this linear formula is a reasonable approximation.

In the experiments we compare all combinations of the two ways to query data sources (see Section 3.5) – using remote NN queries (*RNNQ*) and using remote window queries (*RWQ*) – with the four methods to get an initial query range estimate (see Table 2) – zero (*Zero*), object density (*ObjDens*), object

[3] A server's distance refers to the distance of the closest point on the boundary of its service area.

Table 3. Characteristics of the experiments

	min	max	avg
Data space	Germany (approx. 878 by 610 km)		
Number of data sources	10,000		
Service area size (logarithm of the size is uniformly distributed)	101.0 m^2	225.3 km^2	75.5 km^2
Number of data sources having overlapping service areas	0	10	2.0
Total coverage of data space	74.34 %		
Per query part of a data source's response time (constant per data source, randomly assigned using a negative exponential distribution)	10 ms	1000 ms	100.0 ms
Per object part of a data source's response time (constant per data source, randomly assigned using a negative exponential distribution)	0.3 ms	10 ms	1.0 ms
Number of objects (objects are uniformly distributed across the data space)	1,000,000		
Number of objects per data source	20	452	100.0
Number of nearest neighbors (k)	1	1024	--
Number of threads	1-32, 25%-100%, 1+log, 2*log		

count (*ObjCnt*), and maximum (*Max*). We split the variants in two groups: the *RNNQ*-based and the *RWQ*-based ones. We analyze the behavior of the methods in each group and compare the best method of both groups. *RWQ_ObjDens* and *RWQ_ObjCnt* represent our adaptations to the algorithms introduced in [16]. *RNNQ_ObjCnt* simulates the algorithms presented in [23], while *RNNQ_Max* and *RWQ_Max* simulate the behavior of the other approaches from literature discussed in Section 2.4.

We use four metrics to assess the performance of the algorithm: response time, effort, iterations, and cost profit ratio. Response time is the sum of the response times of the queried data sources in the longest running thread. We assume that the processing effort within the federation component can be neglected as long as the parameter *k* is small. The effort metric[4] quantifies the resource consumption. It is the weighted sum of the queried data sources and all objects retrieved:

- $\text{effort} = (\text{avgPerQueryPartOfResponseTime} \times \text{numberOfDataSourcesQueried}) + (\text{avgPerObjectPartOfResponseTime} \times \text{numberOfObjectsRetrieved})$

Relative response times and efforts are scaled to be multiples of the minimum value for a particular *k*. The number of iterations of the core loop is equivalent to the number of queries to the spatial directory. The cost profit ratio is the product of relative response time and relative effort:

- $\text{cost} = \text{relativeEffort}$
- $\text{profit} = \dfrac{1}{\text{relativeResponseTime}}$
  (less response time means more profit)
- $\dfrac{\text{cost}}{\text{profit}} = \text{relativeEffort} \times \text{relativeResponseTime}$
  (lower is better)

[4] See Table 3 for the values of avgPerQueryPartOfResponseTime and avgPerObjectPartOfResponseTime.

4.1 Experiment 1: Varying Query Size

In the first experiment we vary *k* between 1 and 1024 to show how the proposed algorithm behaves with different query sizes. We use *1+log* threads in the QUERYSERVERS function (see next section for an explanation). This already is an optimization to the approaches from literature which query all sources concurrently. The results of this experiment are shown in Figures 5 and 6.

In Figure 5 the response time and the effort scale better than linearly when *k* increases. The number of iterations of the core loop is little affected by *k*. The variants using the *ObjCnt* and *Max* methods actually do find all result objects in the first iteration. *RWQ_Zero* has problems in finding enough objects to derive a reliable density estimate when *k* is large leading to more iterations.

Figure 6 clearly shows that (for *RNNQ* and *RWQ*) *ObjDens* is the only choice for $k \leq 64$ (up to 23% faster, up to 47% less effort than *ObjCnt*, see arrows in Figure 6). Only for larger *k*, *ObjCnt* and finally *Max* are faster, but *ObjDens* still has a lower effort. *Zero* is second place for small *k*. For larger *k* the performance degrades due to the following effect: initially only few objects get found leading to a large increase of the query range resulting in many data sources to query in the next iteration. The *ObjCnt* and *Max* methods have high fixed costs independent of *k* (see also the effort diagram in Figure 5). For small *k* the effort of the *Max* method exceeds the other methods by orders of magnitude. *ObjCnt* and *Max* consider many data sources as relevant leading to many queries (= high effort) and to an increased probability of having to query a slow data source (= long response time). Only for large *k* the other methods have similarly high costs, so that effort and response time of the *ObjCnt* method get competitive.

For $k \leq 128$, *RNNQ* is clearly faster (up to 29%) and less resource consuming (up to 29%) than *RWQ*. Only for larger *k* *RWQ* slightly surpasses *RNNQ* as it profits from the uniform distribution of the objects and retrieves less unnecessary candidate objects.

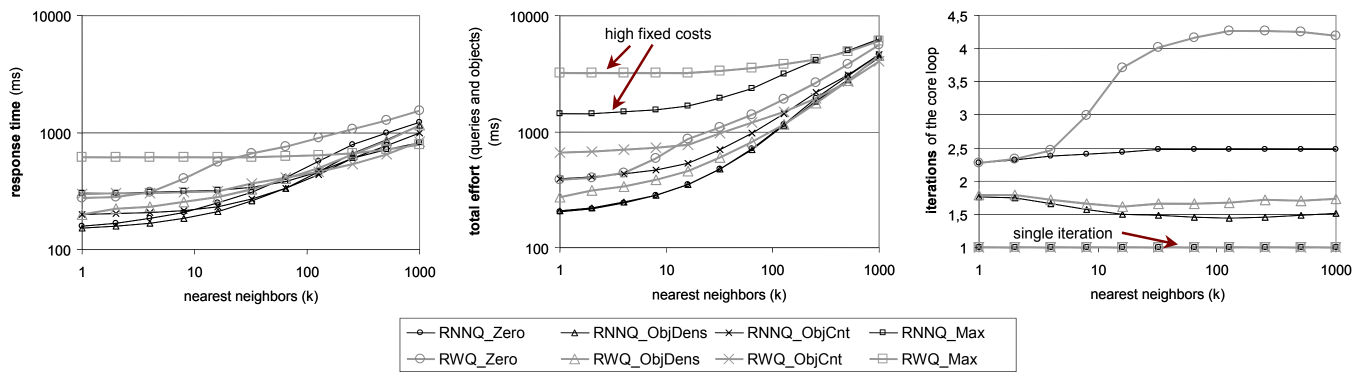

Figure 5. Scalability of our algorithm with increasing *k* (logarithmic x and y axes)

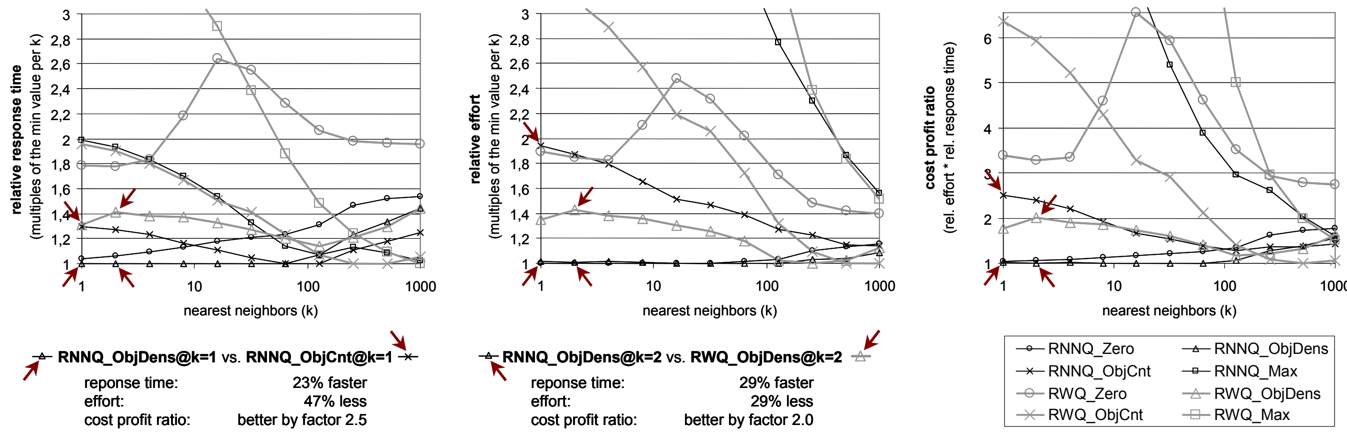

Figure 6. Ranking of the variants with increasing *k* (using *1+log* threads); (the arrows indicate particular points of interest)

4.2 Experiment 2: Varying Parallelism

In the second experiment we vary the number of threads in the QUERYSERVERS function to show the influence of parallel queries to remote data sources on the performance of the proposed algorithm. The number of threads is either statically fixed between 1 and 32, or depends dynamically on the size of the server queue: Then, the thread count is either a fraction of the server queue size (*25%, 33%, 50%, 100%*) or proportional to its logarithm (*2*log, 1+log*):

- 2*log = $2 \times \log_2(\text{ServersPQ.size})$
- 1+log = $1 + \log_2(\text{ServersPQ.size})$

We set *k* to 64. The most promising approach from literature is comparable to the *RNNQ_ObjCnt* variant querying all relevant data sources concurrently (*100%*). This will serve as the reference behavior for the discussion of our new approach. The results of this experiment are shown in Figure 7.

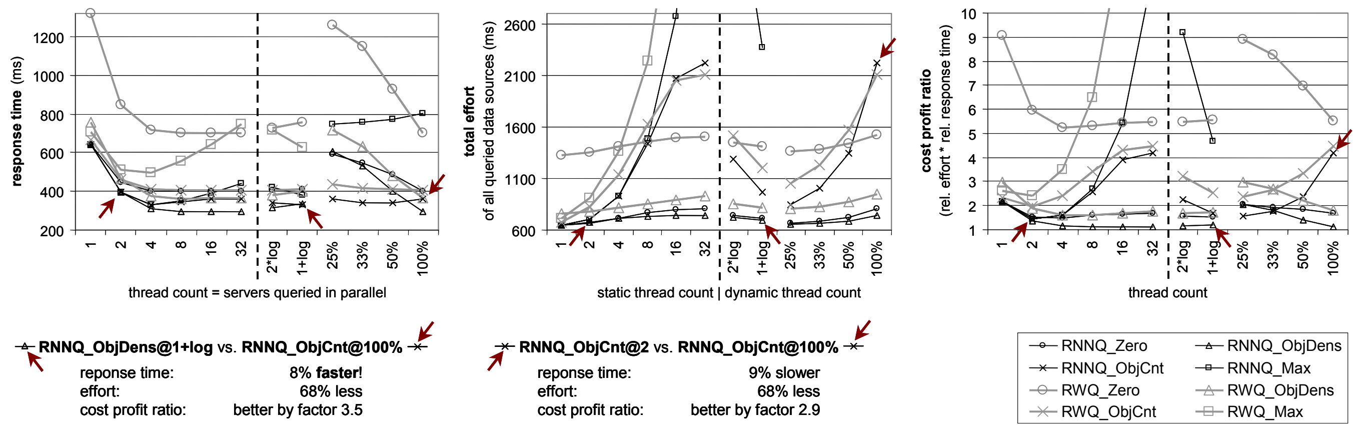

Figure 7. Performance of our algorithm with varying number of threads (*k* = 64); (the arrows indicate particular points of interest)

Obviously, using a single thread leads to long response times and minimal resource consumption, while querying all data sources at the same time (*100%*) leads to minimal response times and maximal resource consumption. Calculating the thread count from the logarithm of the server count (*1+log*) achieves the best tradeoff. Generally, the response time is lower the more threads are used. Only with *Max* it gets worse, because of the higher probability of querying a slow data source.

When using the *ObjDens* method few data sources are queried, and thus the effort is only slightly affected by the thread count. The reverse holds for the *ObjCnt* method (queries many data sources by nature, and all of them in only one iteration) and for the *Zero* method (range-over-estimation-effect). Using the *Max* method with many threads leads to a horrendously high effort. The *RNNQ*-based and the *RWQ*-based variants show a similar behavior when the thread count is varied. When considering individual methods, *RNNQ* always performs better than the *RWQ* counterpart for $k = 64$.

To optimize the cost profit ratio, Figure 7 suggests to use one of the following settings:

- **2 threads**: Yields a low effort and slightly suboptimal response time. However, the static number 2 depends on k and the system environment. For example, 4 threads are better for $k = 256$.
- **1+log Threads**: Yields a close to optimal response time and a slightly higher effort. This setting is independent of k. It yields the best result of the dynamic settings.
- **25%**: Is only advisable for the *ObjCnt* method. Yields a low effort and slightly suboptimal response time independent of k. For *Zero* and *ObjDens* this setting effectively means to query the data sources sequentially.
- **100%**: Is only advisable for the *ObjDens* method, as few data sources are queried. Yields the best response time at a remarkably low effort. The cost profit ratio is even slightly better than for the *1+log* setting (only in conjunction with the *ObjDens* method).

4.3 Summary

Our experiments show that *ObjDens* is the best method to get an initial query range estimate, as it outperforms all other variants for $k \leq 64$ (cost profit ratio up to 2.5 times better) and still has the best effort for larger k while only depending on relatively easy to get object density statistics. *RNNQ* is preferable when offered by the data sources, as it is more robust for non-uniform object distributions. Yet, using *RWQ* for $k \leq 128$ involves at most 29% response time and effort degradation. The *1+log* threads setting achieves the best trade off between response time (at most 12% above the minimum at *100%* threads) and effort (at most 79% above the minimum at *1* thread) and works equally well for all variants. Combining all parameters, the *RNNQ_ObjDens* method using *1+log* threads outperforms the most promising approach from literature (represented by *RNNQ_ObjCnt* using *100%* threads) by a factor of 3.5. If we take into account that for many applications realistic k values are in the order of 10, and that response times in the order of less than two seconds are well acceptable, then our evaluation clearly shows that *ObjDens* and also *Zero* are the preferred methods. Also, we have plenty of room to trade in some response time in order to minimize the overall resource consumption, which is crucial for a large-scale system.

5. CONCLUSION

In this paper we address the problem of processing nearest neighbor (NN) queries in a federated environment of spatial data sources that are loosely coupled over the internet. We present an algorithm to process such federated nearest neighbor queries (FNNQs), discuss variants of it, and report on experimental results. We pay special attention to minimizing the number of queried data sources by dynamically enlarging and shrinking the set of relevant sources. We show that extending the set of parameters – using moderate parallelism, and determining the relevant data sources by using service areas and a global object density estimate – leads to considerable performance improvements. Parallelization serves as a controlling element to trade response time for resource consumption. The experiments show that the algorithm using remote NN queries, object density statistics, and *1+log* threads is the best variant to process FNNQs. Previous approaches from literature can be employed to implement the local processing of a NN query at a remote data source. Nevertheless, the proposed algorithm can also do without remote NN queries and use remote window queries instead. The NN query problem in a federated environment truly exceeds present knowledge on NN queries.

Yet, some future work is conceivable:

- Increase the precision of determining the relevant sources: consider size and placement of service areas, combine the object density-based and the object count-based method, exploit additional metadata, etc.
- Derivation of analytical formulas describing the behavior of each variant, so that cost-based optimizers can dynamically choose the right one.

Still, all those results would not change the reported achievements.

6. ACKNOWLEDGEMENTS

The Nexus project was funded 1999-2002 by the German Research Association (DFG) under grant 200989 and is now continued as Center of Excellence (SFB) 627.

7. REFERENCES

[1] S. Arya: *Nearest Neighbor Searching and Applications*, Ph.D. thesis, Dept. of Computer Science, University of Maryland, College Park, MD, USA, 1995

[2] S. Berchtold, C. Böhm, B. Braunmüller, D. A. Keim, H.-P. Kriegel: *Fast Parallel Similarity Search in Multimedia Databases*, SIGMOD 1997, Proc. ACM SIGMOD International Conference on Management of Data, May 1997, Tucson, Arizona, USA, pp. 1-12

[3] S. Berchtold, B. Ertl, D. A. Keim, H.-P. Kriegel, T. Seidl: *Fast Nearest Neighbor Search in High-dimensional Space*, Proc. of the 14th Intl. Conf. on Data Engineering (ICDE'98), Orlando, Florida, Feb 1998

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger: *The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles,* Proc. of the 1990 ACM SIGMOD Intl. Conf. on Management of Data, Atlantic City, New Jersey, USA, ACM Press, 1990, pp. 322-331

[5] K. L. Cheung, A.W. Fu: *Enhanced Nearest Neighbour Search on the R-tree*, SIGMOD Record, vol. 27, no. 3, pp. 16-21, 1998

[6] P. Ciaccia, A. Nanni, M. Patella: *A Query-sensitive Cost Model for Similarity Queries with M-tree*, Proc. of the 10th Australasian Database Conference (ADC'99), Auckland, New Zealand, Jan 1999, pp. 65-76

[7] H. Ferhatosmanoglu, D. Agrawal, A. El Abbadi: *Concentric Hyperspaces and Disk Allocation for Fast Parallel Range Searching*, Proceedings of the 15th International Conference on Data Engineering, ICDE 1999, March 1999, Sydney, Austrialia, pp. 608-615

[8] H. Ferhatosmanoglu, D. Agrawal, A. El Abbadi: *Optimal Partitioning for Efficient I/O in Spatial Databases*, Euro-Par 2001: Parallel Processing: 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, LNCS 2150 / 2001, pp. 889-900

[9] C. Faloutsos, P. Bhagwat: *Declustering Using Fractals*, Proc. of the 2nd Intl. Conf. on Parallel and Distributed Information Systems (PDIS 1993), San Diego, CA, USA, January 1993, pp. 18-25

[10] R. Fagin: *Combining Fuzzy Information from Multiple Systems*, Proc. of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1996, June 1996, Montreal, Canada, pp. 216-226

[11] The Federal Geographic Data Committee: *The Clearinghouse*, http://www.fgdc.gov/clearinghouse/clearinghouse.html

[12] G. R. Hjaltason, H. Samet: *Ranking in Spatial Databases*, Proc. of the 4th Symp. on Spatial Databases, Portland, Maine, USA, Aug. 1995, LNCS 951, pp. 83-95

[13] G. R. Hjaltason, H. Samet: *Distance Browsing in Spatial Databases*, ACM Transactions on Database Systems, vol. 24, no. 2, pp. 265-318, 1999

[14] N. Koudas, C. Faloutsos, I. Kamel: *Declustering Spatial Databases on a Multi-Computer Architecture*, Proc. of the 5th International Conference on Extending Database Technology, EDBT'96, Avignon, France, March 1996, pp. 592-614

[15] Y.-l. Lo, K. A. Hua, H. C. Young: *A General Multidimensional Data Allocation Method for Multicomputer Database Systems*, Database and Expert Systems Applications, 8th International Conference, DEXA '97, Toulouse, France, September 1-5, 1997, pp. 357-366

[16] Dan-Zhou Liu, Ee-Peng Lim, Wee-Keong Ng: *Efficient k Nearest Neighbor Queries on Remote Spatial Databases Using Range Estimation*, Proc. of the 14th Intl. Conf. on Scientific and Statistical Database Management (SSDBM'02), Edinburgh, Scotland, July 2002, pp. 121-130

[17] D. Nicklas, M. Großmann, T. Schwarz, S. Volz, B. Mitschang: *A Model-Based, Open Architecture for Mobile, Spatially Aware Applications*, Proc. of the 7th Intl. Symp. on Spatial and Temporal Databases (SSTD01), Los Angeles, LNCS 2121, 2001, pp. 117-135

[18] Open GIS Consortium: *OpenGIS Simple Features Specification for SQL, Revision 1.1*, Open GIS Consortium (OGC), May 1999, http://www.opengis.org/docs/99-049.pdf

[19] Open GIS Consortium: *The OpenGIS Abstract Specification - Topic 13: Catalog Services*, Open GIS Consortium (OGC), March 1999, http://www.opengis.org/docs/99-113.pdf

[20] Open GIS Consortium: *OpenGIS Catalog Services Specification*, Open GIS Consortium (OGC), Dec. 2002, http://www.opengis.org/docs/02-087r3.pdf

[21] Open GIS Consortium: *OWS1.2 UDDI Experiment*, Open GIS Consortium (OGC), Jan 2003, http://www.opengis.org/docs/03-028.pdf

[22] S. Prabhakar, K. A. S. Abdel-Ghaffar, D. Agrawal, A. El Abbadi: *Cyclic Allocation of Two-Dimensional Data*, Proc. of the 14th Intl. Conf. on Data Engineering, ICDE 1998, February 1998, Orlando, Florida, USA, pp. 94-101

[23] A. Papadopoulos, Y. Manolopoulos: *Parallel Processing of Nearest Neighbor Queries in Declustered Spatial Data*, Proc. of the fourth ACM workshop on Advances in Geographic Information Systems, ACM-GIS 1996, Rockville, Maryland, United States, pp. 35-43

[24] A. Papadopoulos, Y. Manolopoulos: *Similarity Query Processing Using Disk Arrays*, SIGMOD 1998, Proc. ACM SIGMOD Intl. Conf. on Management of Data, June 1998, Seattle, Washington, USA, pp. 225-236

[25] A. Papadopoulos, Y. Manolopoulos: *Distributed Processing of Similarity Queries*, Distributed and Parallel Databases, Volume 9, Issue 1, January 2001, pp. 67 - 92

[26] H. Pinto, N. V. Boas, R. José, *Using a private UDDI for publishing location-based information to mobile users*, ICCC/IFIP 7th Intl. Conf. on Electronic Publishing (ElPub2003), Guimarães, Portugal, June 2003

[27] N. Roussopoulos, S. Kelley, F. Vincent: *Nearest Neighbor Queries*, Proc. of the 1995 ACM-SIGMOD Intl. Conf. on Management of Data, San Jose, CA, USA, May 1995, pp. 71-79

[28] UDDI: *The UDDI Technical White Paper*, UDDI.org, Sept. 2000, http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf

[29] C. Yu, B.C. Ooi, K.-L. Tan, H.V. Jagadish: *Indexing the Distance: An Efficient Method to KNN Processing*, Proc. of the 27th VLDB Conf., Roma, Italy, 2001