# Cloud-Assisted Spatio-Textual $k$ Nearest Neighbor Joins in Sensor Networks

Mingyang Yang[1], Long Zheng[1], Yanchao Lu[1], Minyi Guo[1], Jie Li[2]
Department of Computer Science and Engineering, Shanghai Jiao Tong University
[1]{ymycser, longzheng, chzblych, myguo}@sjtu.edu.cn, [2]lijie@cs.sjtu.edu.cn

***Abstract*—$k$ nearest neighbors ($k$NN) query is an important problem in a variety of sensor network applications. Traditionally, we handle this problem with a single query processing approach, which just considers the location information. It usually neglects the other information such as temperature, humidity, pressure, etc. In order to overcome the defect of the traditional approaches, we investigate the problem from a new perspective and desire to solve a more interesting problem called spatio-textual $k$ nearest neighbor join (ST-$k$NNJ). It searches text-similar and $k$-nearest sensors to a query set containing more than one query point. With the help of cloud computing, ST-$k$NNJ can be processed in distributed computational environment to gain better processing capability and response efficiency. In this paper, we generalize the problem of ST-$k$NNJ and propose our approaches to it. And we can deal with large-scale data when using MapReduce framework. Evaluation results show that our approach achieve better performance in comparison with the naive approach.**

***Keywords*—Sensor Networks, Cloud Computing, Distributed Computing, MapReduce, k Nearest Neighbor Join***

## I. Introduction

Wireless sensor networks (WSNs) commonly consist of a large number of distributed sensors which are capable of sensing, communicating and computing. After deployed over wide geographical areas, WSNs are used to monitor environment and collect data[6]. Typically, these sensors can obtain various types of data, including geographical location, temperature, humidity, illumination and pressure. With the wide use of WSNs, a variety of applications, such as military and environmental monitoring, intelligent transportation, and location-based services (LBS), have significantly developed.

In these applications, spatial queries are very important as sensed data are often geographically distributed[23]. Particularly, $k$ nearest neighbor ($k$NN) query is a common operation relevant to WSNs. It can retrieve the information of k nearest neighbors ($k$NNs) for a given query point. Applications usually require information from certain number (e.g. $k$) of sensors that are closest to a specified location. For instance, soldiers can issue $k$NN queries to obtain $k$ nearest tanks from them in battlefields.

Recent years, many works[26], [23], [28], [22], [6], [4], [9] have focused on $k$NN queries in networks including sensor networks, ad hoc networks, peer-to-peer networks. But their approaches just handle a $k$NN query at a time. Nowadays, WSNs based applications have become much more popular and users usually launch a number of queries at the same time. If we just handle these queries one by one, the response efficiency is relatively low. It is very necessary to gather these queries and handling them in batches. We can organize queries to a query set and perform $k$NN join query on the query set and the sensor set. $k$NN join searches $k$ nearest neighbors of all the query points from the sensor set, which makes handling multiple queries simultaneously possible. Moreover, previous works mentioned above only process spatial information. As we know, the deployed sensors collect not only spatial information, but also temperature, humidity, pressure, etc. that are stored in the form of text or string. In some cases, users wants to get specific query results according to all of these information. Text information between $k$NNs and the query point should be similar to a certain degree, which means, some sensors spatially closer to the query point may not be included in the query result as their text information do not satisfy the query condition. If we combine this kind of query with $k$NN join, we can issue a novel and powerful query called spatio-textual $k$ nearest neighbor join (ST-$k$NNJ).

However, if we launch ST-$k$NNJ query in networks, like previous work[23], [22], [6], [9], called in-network processing techniques, energy consumption and communication overhead will increase while energy budget and network bandwidth is limited in sensors. This may accelerate sensors aging. Some other work[26], [8], [29] transfer query processing to centralized database servers. With the emergence of cloud computing, processing ST-$k$NNJ query in a distributed computational environment is a natural consideration as the sensed data become extremely large. This can improve processing capability and response efficiency. We can process the collected data in the clouds, and then provide query applications or services for users, like Software-as-a-Service (SaaS). MapReduce[5] is a programming framework for processing large-scale datasets by exploiting the parallelism among a cluster of computing nodes, which typically consists of a number of commodity machines. As its simplicity, flexibility, fault tolerance and scalability, MapReduce has gained increasing support from both industry and academia in the past few years, and become one of the most widely used frameworks for parallel and distributed processing of large data nowadays.

In this paper, we study a new problem, called cloud-assisted spatio-textual $k$ nearest neighbor join (ST-$k$NNJ) in sensor networks. ST-$k$NNJ finds each query point $r$ of a query set $R$ and its $k$-nearest sensors $\{s_1, s_2,..., s_k\}$ from a sensor set $S$. Both the query points and the sensors contain location and text information. These $k$ sensors satisfy the following two constraints: they are the top-$k$ nearest neighbors to $r$ in space and meanwhile their text must be similar to $r$'s. Besides, we solve the problem of ST-$k$NNJ in MapReduce framework to deal with large-scale data in the clouds. We first present a naive approach using block nested loops. It retrieves query

INISCom 2015, March 02-04, Tokyo, Japan
Copyright © 2015 ICST
DOI 10.4108/icst.iniscom.2015.258321

results by traversing the query set and the sensor set. Then we propose our improved approach which searches text-similar intermediate results and finds $k$NNs from them to obtain the final query results. To our best knowledge, it is the first work investigating ST-$k$NNJ query for sensor networks. Due to the space limitation of this paper, we evaluate the performance of this two ST-$k$NNJ query processing approaches with extensive datasets.

Our main contributions are summarized as follows:

- We study a new problem called $spatio-textual\ k\ nearest\ neighbor\ joins\ in\ sensor\ networks$ and formalize the notion of it.
- We propose a text similarity based approach which first finds text-similar pairs and then gets the query results using spatial information.
- We have conducted comprehensive experiments on large-scale datasets to evaluate our approach. Experiment results show that our approach achieve better performance in comparison with the naive approach.

The rest of this paper is organized as follows. We formulate the ST-$k$NNJ problem in Section II. Some background knowledge is described in Section III. In Section IV, we present the naive approach and our ST-$k$NNJ approaches in detail. In Section V, experiment results are reported. We survey the related work in Section VI and make a conclusion in Section VII.

## II. Problem Formulation

We formulate the problem of spatio-textual $k$ nearest neighbor join (ST-$k$NNJ) in sensor networks. Consider a query set $R$ = $\{r_1,\ r_2, ...,\ r_u\}$ and a sensor set $S$ = $\{s_1,\ s_2, ...,\ s_v\}$, where $|R| = u$ and $|S| = v$. Each point $r$ (the same to $s$) contains 2-dimensional coordinates $C_r$ = $(c_{r,x},\ c_{r,y})$ and a text $T_r$. Given a fixed number $k$ and a text similarity threshold $\tau$, for each point $r \in R$, ST-$k$NNJ finds $k$ sensors $\{s_1,\ s_2, ...,\ s_k\}$ from $S$ that satisfy the following two constraints:

- Textual constraint: The text similarities between these $k$ sensors and $r$ are not less than $\tau$.
- Spatial constraint: These $k$ sensors are the top $k$ nearest neighbors to $r$ in terms of some spatial distance metric.

In this work, we use edit distance to evaluate the text similarity and Euclidean distance to evaluate the spatial distance respectively. Furthermore, as we carry out the computation process in the cloud servers, all the sensed data including location and text information are stored in cloud servers and get refreshed periodically according to specific application requirements.

## III. Preliminaries

In this section, we introduce some background knowledge and techniques used in this work.

### A. MapReduce

MapReduce[5] is a popular programming framework for parallel and distributed processing large-scale datasets using shared-nothing clusters. A typical MapReduce program consists of a pair of user-defined $map$ and $reduce$ functions. Invoked for every record in the input datasets, the map function inputs a key-value pair and produces a list of intermediate key-values pairs. Then these intermediate pairs are partitioned according to the keys. For each partition, all the values associated with the same intermediate key are sorted and organized into a list. This procedure is called shuffle. The reduce function gets sorted data from the appropriate partition and outputs the final result which is typically a list of key-value pairs. The computational process can be described conceptually as follows: map(k1, v1) $\rightarrow$ list(k2, v2) and reduce(k2, list(v2)) $\rightarrow$ list(k3, v3).

Hadoop is an open source implementation of MapReduce, which is popular in the open source community. In Hadoop, data are stored in Hadoop distributed file system (HDFS). HDFS contains multiple slave nodes called DataNodes which store data in chunks and a master node called NameNode which stores meta-data and monitors DataNodes. Hadoop runtime system launches two kinds of processes, called JobTracker and TaskTracker, to handle MapReduce programs. In Hadoop, a MapReduce program is called a job. JobTracker splits a submitted job into map tasks and reduce tasks, and schedules them to TaskTrackers. For a map task, TaskTracker fetches a data chunk from DataNode and runs the map function. After all the map tasks are completed, Hadoop runtime system partitions and sorts the intermediate data. This procedure is called shuffle. Then multiple TaskTrackers for reduce tasks are launched to run the reduce function for producing the final result.

### B. Euclidean Distance

Commonly, we use Euclidean distance to measure the distance between two spatial points in Euclidean space. In Cartesian coordinates, if $r$ = $(c_{r,1},\ c_{r,2}, ...,\ c_{r,n})$ and $s$ = $(c_{s,1},\ c_{s,2}, ...,\ c_{s,n})$ are two objects in Euclidean n-space, their Euclidean distance $d(r,s)$ is defined as:

$$d(r,s) = \sqrt{\sum_{i=1}^{n} (c_{r,i} - c_{s,i})^2}$$

### C. Edit Distance

In this paper, we use edit distance to evaluate the similarity between texts. Formally, given texts $T_1$ and $T_2$, their edit distance, denoted by $ed(T_1, T_2)$, is the minimum number of single-character edit operations (i.e., insertion, deletion, and substitution) needed to transform $T_1$ to $T_2$. For example, consider $T_1$ = "sensor" and $T_2$ = "sense", $ed(T_1, T_2)$ is 2.

## IV. ST-$k$NN Join

In this section, we introduce two approaches to deal with ST-$k$NNJ. A naive approach is to perform a brute-force search of all points in query set $R$ and sensor set $S$. Specifically, for each query point $r \in R$, we scan all sensors in $S$ to get the final exact results, which means nested loop scans are performed on

$R$ and $S$. Besides, we present an improved approach that is more query-efficient than the former naive one. This approach is based on the state-of-the-art $filter\text{-}and\text{-}refine$ framework.

### A. Nested Loop Approach

To perform nested loop scans on $R$ and $S$ in MapReduce framework, a straightforward idea is, map tasks used for data partitioning and reduce tasks used for nested loop scanning. Data partitioning can distribute data to all cloud servers where reduce tasks perform nested loop scanning. If map tasks just input $R$ and partition it into multiple splits, then each reduce task launched by cloud server gets one split of $R$ and must input all sensed data of $S$ to finish the computation. This strategy needs only a single MapReduce job. In many applications, the size of $S$ is usually much larger than the size of $R$. Obviously, this strategy may increase the system I/O burden and resource consumption.

An improvement is to use block nested loop methodology mentioned in [27]. This new approach adopted block nested loop methodology consists of two MapReduce jobs. We denote this approach as ST-BNLJ and it is described as follow:

1) **Block Join Job**:

- **Map Phase**: After reading $R$ (or $S$), map tasks perform a linear scan on $R$ (or $S$) and put every $|R|/n$ (or $|S|/n$) points into one block. Then $R$ and $S$ are both partitioned into $n$ equal-sized blocks. Every possible pair of blocks (one from $R$ and another from $S$) is delegated into a bucket. So this phase produces a total of $n^2$ buckets.
- **Reduce Phase**: $n^2$ reduce tasks are launched in cloud servers. Each reduce task fetches a bucket produced in the map phase and performs a block nested loop join between blocks of $R$ and $S$ in that bucket. That is, by using a nested loop, for each query point $r$ in the block of $R$, we scan all sensors in the block of $S$ to check textual constraint and spatial constraint to obtain local $k$NNs. We can calculate edit distance with dynamic programming to check textual constraint. A max heap of Euclidean distances can be used to get $k$NNs. The output is $r$ and its local $k$NNs.

Note in the first job, each query point $r \in R$ is partitioned into one block of $R$ and replicated in $n$ buckets (one for each of the $n$ blocks of $S$). We can see that, when the first job is finished, each query point $r \in R$ has $n$ local $k$NNs. Hence, we use another job to merge these local $k$NNs to get global $k$NNs for $r$.

2) **Merging Job**:

- **Map Phase**: Map tasks read all outputs of the first job and transfer each query point $r \in R$ and its $nk$ $k$NN candidates to a reduce task. This can be done by using the unique ID (e.g. $pointID_r$) of each query point $r$ as output key of map tasks.
- **Reduce Phase**: A reduce task retrieves each query point $r \in R$ and its $nk$ $k$NN candidates, and then sort these candidates in ascending order of $d(r, candidate)$. Finally we get the top-$k$ results (i.e. global $k$NNs) for each query point $r$.

### B. $filter\text{-}and\text{-}refine$ based Approach

When the number of query points or deployed sensors increases, the efficiency of nested loop approach may be affected due to its brute-force search. In order to overcome this defect, we propose an improved approach. From the problem definition, we can know that, sensors from the final results must first satisfy the textual constraint before comparing spatial distances. Hence we can explore the potential of text information and use them to prune useless data. Inspired by the work in [11], we adopt the the state-of-the-art $filter\text{-}and\text{-}refine$ framework in this approach.

Consider a query point $r \in R$ and a sensor $s \in S$, if they are not text-similar, then $s$ can be pruned. We can use an inverted index to filter out text-similar candidates and then verify them to get final results. In inverted indices, texts are treated as index keys and points containing the corresponding text are index values.

As mentioned above, we use edit distance to evaluate the text similarity between texts. A technique based on $q$-grams and a $q$-gram counting argument have been proposed to identify candidate texts within a small edit distance from a query text fast[10]. Let $\Sigma$ be an alphabet. For a text $T$ in $\Sigma$, we use a sliding window of length $q$ over the characters of it to produce its $q$-grams. In particular, the beginning and the end of $T$ have less than $q$ characters. Special characters which are not in the $\Sigma$, such as # and \$, are introduced to deal with them. Then $T$ is extended by prefixing it with $q$ - 1 occurrences of # and suffixing it with $q$ - 1 occurrences of \$, so that the $q$-grams in the beginning and the end of $T$ have exactly $q$ characters. For instance, consider $q$ = 3 and $T$ = ”network”, the $q$-grams of $T$ are {##n, #ne, net, etw, two, wor, ork, rk\$, k\$\$}. Texts within a small edit distance will share a large number of $q$–grams. Let $G_T$ be the set of $q$–grams of text $T$. From [10], we have the following conclusion:

For texts $T_1$ and $T_2$ of length $|T_1|$ and $|T_2|$, if $ed(T_1, T_2) = \tau$, then $|G_{T_1} \cap G_{T_2}| \geq max(|T_1|, |T_2|) - 1 - (\tau - 1) \times q$ $(*)$.

So we can first construct inverted indices based on $q$-grams of texts of sensors. As the sensed data are stored in cloud servers and get refreshed periodically, once the indices are constructed after refreshment, the subsequent queries share the same indices. For $r \in R$, we get its $q$-grams and their corresponding inverted indices. After counting the occurrence of sensors in these indices, the above conclusion is used to get $r$’s text-similar candidates. At last, we verify the filtered candidates by checking textual constraint and spatial constraint. This $filter\text{-}and\text{-}refine$ based approach consists of two MapReduce jobs. The first job is a preprocessing job. It constructs inverted indices for the sensor set $S$. The second job is used for queries. It filters out text-similar sensors for each query point, and refines these candidates to outputs the final query results. We denote this approach as ST-FRJ and it is described as follow:

1) **Preprocessing Job**:

- **Map Phase**: Map tasks input query set $S$. For each sensor $s \in S$, $T_s$ is divided into $|T_s| - q + 1$ $q$-grams. Then map tasks use $(gram, s)$ as output key-value pair.

- **Reduce Phase**: A reduce task retrieves all key-value pairs $(gram, s)$ that share the same $gram$. Hence, $gram$ is used for inverted index key and all these sensors are used for inverted index values. At last, reduce tasks output all inverted indices.

2) **Query Job**:

- **Map Phase**: Map tasks simply read query set $R$ and evenly disperse them to all reduce tasks. This can be done by using $pointID \ mod \ n$ (i.e. number of reduce tasks) as output key and query point as output value.
- **Reduce Phase**: A reduce task fetches a chunk of $R$. For each query point $r$ in this chunk, we first divide $T_r$ into $q$-grams and read the corresponding inverted indices. Then the conclusion mentioned above is used to filter out text-similar candidate sensors. Then we verify the filtered candidates to obtain the final $k$NNs by calculating their edit distance and Euclidean distance.

## V. Experiments

We implement our approaches in Java with Hadoop APIs and execute our experiments on our in-house cluster. The cluster contains 10 computing servers, each of which has one Intel Xeon E5645 2.4GHz CPU, 64GB memory, two 200GB SATA hard disks and Gigabit ethernet. On each server, we install CentOS 6.5 operating system, Java 1.7.0 with 64-Bit server VM, and Hadoop 1.0.4. And we make the following modifications to the default Hadoop configurations: (1) each server is set to run 10 map and 10 reduce tasks; (2) the size of virtual memory for each map and reduce task is set to 2GB; (3) the replication factor is set to 1.

We evaluate the naive approach ST-BNLJ and the improved approach ST-FRJ on simulated query set (denoted as $R$) and sensor set (denoted as $S$). Totally, $R$ contains 32K query points and $S$ contains 320K sensors. Each point of $R$ (or $S$) contains 2-dimensional coordinates (spatial information) and a text (text information) whose length is at least 10 and no more than 20.

In our experiments, we measure the effect of the following parameters to our approaches: (1) gram length $q$; (2) text similarity threshold $\tau$; (3) number of nearest neighbors $k$; (4) size of $R$; (5) size of $S$; (6) number of cores in a single computing server; (7) number of computing servers. Due to the space limitation of this paper, we evaluate performance of our approaches with execution time. By default, we conduct the performance evaluation on $R$ with 1K query points and $S$ with 80K sensors. $q$ is set to 2, $k$ is set to 20, $\tau$ is set to 3, and the number of computing servers is 10.

### A. *Effect of $q$*

From Fig. 1, we observe that the running time of preprocessing job of ST-FRJ increases substantially when the gram length $q$ increases. But the running time of query job of ST-FRJ remains almost the same. It means a short gram (e.g. 2 or 3 in our experiments) is suitable for inverted indices construction. So we choose a short gram in the following experiments. Note that, although ST-FRJ contains two jobs: preprocessing job and query job. But preprocessing job runs only when the sensed data ($S$) get refreshed periodically according to specific application requirements. There is no need to run it when every query request comes. So we mainly focus on the query performance of ST-BNLJ and ST-FRJ.

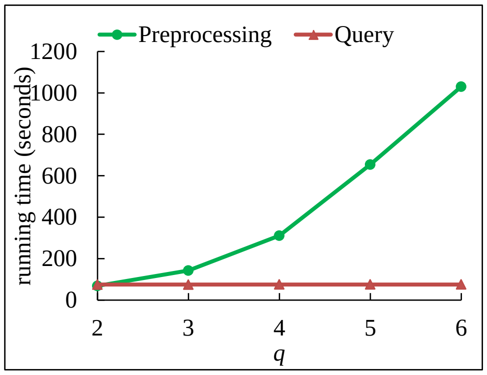


Fig. 1. Effect of Gram Length $q$

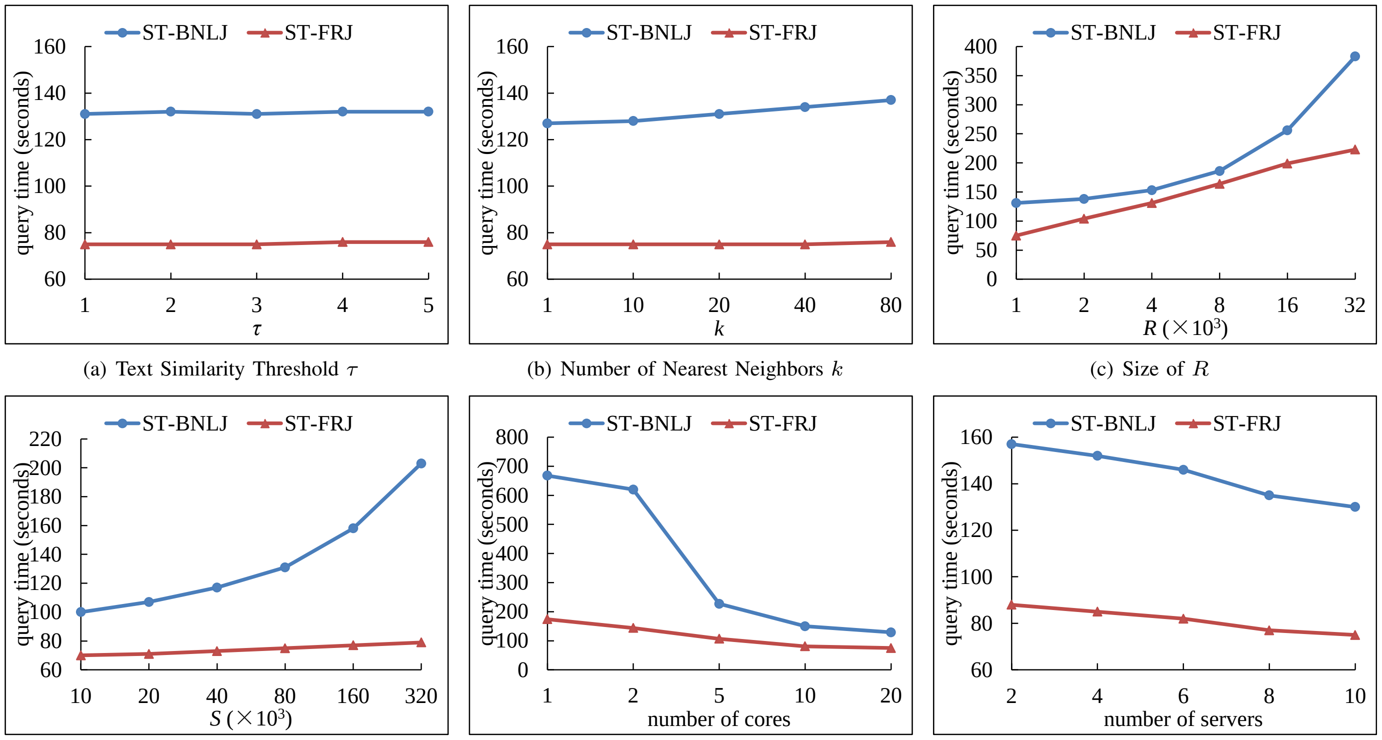
### B. *Effect of $\tau$ and $k$*

From Fig. 2(a), we can see that when text similarity threshold $\tau$ increases, the query time of ST-BNLJ (and ST-FRJ) remains almost the same. In ST-BNLJ, $\tau$ is used to check textual constraint after calculating edit distance in the block join job. In ST-FRJ, $\tau$ is used to filter out candidate sensors after counting the occurrence of sensors in the inverted indices in the query job. Its change dose not incur any transmission overhead for all these approaches. Although a larger $\tau$ may bring more candidates, which may make the max heap adjustment more frequently. But when the cloud severs is powerful enough, this impact can be negligible.

A little difference in Fig. 2(b) is when the number of nearest neighbors $k$ increases, the query time of ST-BNLJ increases slightly while the query time of ST-FRJ remains almost the same. In ST-BNLJ, a larger $k$ will incur more intermediate results (i.e. local $k$NNs). So the transmission overheads between its two jobs increase. In ST-FRJ, $k$'s change dose not incur any transmission overhead. Hence the query time of ST-FRJ is almost the same.But no matter how $\tau$ and $k$ vary, ST-FRJ always achieves better query performance than ST-BNLJ.

### C. *Effect of $R$ and $S$*

From Fig. 2(c) and Fig. 2(d), we can observe that when the sizes of query set $R$ and sensor set $S$ increase, the query time of both ST-BNLJ and ST-FRJ increases. But their growth rates are different. The query time of ST-FRJ gets linear growth, while the query time of ST-BNLJ grows faster. That's because ST-BNLJ adopts a brute-force search (i.e. nested loop seach) on $R$ and $S$. However, ST-FRJ can usually prune a lot of useless data by utilizing inverted indices to reduce the number of candidates. In the worst case, all sensors become text-similar candidates and ST-FRJ may get the same growth rate of query time as ST-BNLJ. But on average, it has better performance than ST-BNLJ and its query time is always less than ST-BNLJ.



(a) Text Similarity Threshold $\tau$ (b) Number of Nearest Neighbors $k$ (c) Size of $R$

(d) Size of $S$ (e) Number of Computing Cores $U$ (f) Number of Computing Servers $N$

Fig. 2. Effects of $\tau$, $k$, $R$, $S$, $U$ & $N$

### D. *Effect of Computing Cores and Servers*

We first measure the effect of computing cores in a single cloud server. This can be done by configuring Hadoop to run in a single cloud server (i.e. it is master and single slave at the same time.). We can change the Hadoop configuration by tuning the number of map tasks (or reduce tasks) running simultaneously. In this way, we can control the number of computing cores. From Fig. 2(e), we can see that when the computing cores increases, the query time of ST-BNLJ and ST-FRJ decreases, which means our approaches are compute-intensive. ST-BNLJ gets a larger decrement when the computing cores increases from a small number.

As our approaches are compute-intensive, we extend the experiments to more than one cloud server. We enlarge the number of computing servers from 2 to 10 in order to see its effect. From Fig. 2(f), a obvious trend is that with the number of computing servers increasing, the query time of ST-BNLJ and ST-FRJ decreases. It means more computing servers can bring more query performance promotion. However, no matter how the number of computing servers (or cores) varies, ST-FRJ always gains better query performance than ST-BNLJ.

## VI. Related Work

Many studies have focused on $k$NN queries in WSNs. In [8], [29], fixed communication infrastructure and centralized database servers are required, and query processing and data storage rely on centralized database servers. Some approaches[3], [7] delegate partial query monitoring tasks to the mobile user, in order to reduce communication overhead. Existing in-network $k$NN query processing approaches can be classified into two categories: infrastructure-based approaches and infrastructure-free approaches. The former ones rely on a network infrastructure for query propagation and processing, as in[16], [17], [23]. The latter ones propagate $k$NN queries along some well-devised itineraries to collect data, and some infrastructure-free $k$NN query processing approaches based on itinerary structures have been proposed in [23], [18], [19], [6]. In [23], the authors proposed three methods for $k$NN query processing in location-aware sensor networks. GRT and KBT are based on tree infrastructure while IKNN is an infrastructure-free approach. DIKNN, another infrastructure-free approach proposed in [19], uses routing, $k$NN boundary estimation, and query dissemination to get query results. In [6], an approach called PCIKNN is proposed which calculates the network density during the traversal of the itineraries and partitions the search range to reduce the response time of query execution.

ST-$k$NNJ can be considered as a combination of spatial $k$NN join and text similarity join. In traditional database area, there have been many works that study the $k$NN join and text similarity join, and many approaches have been developed to solve this two problems. In [20], [25], [24], $k$NN join is performed on a single and centralized machine. When the size of datasets becomes extremely large or the data become multi-dimensional, the performance of the single machine becomes the bottleneck to solve this problem. To cope with large dataset or multi-dimensional data, several work[27], [13] is proposed using MapReduce framework. The approach in [27] computes approximate results while the approach in [13] computes exact results.

Li et al. have given a conclusion of string similarity joins in [11]. Existing studies addressing this problem can be mainly classified into two categories: $filter\text{-}and\text{-}refine$ framework and $trie$ based framework. The first one contains a $filter$ step and a $refine$ step. The $filter$ step generates signatures for each strings and uses the signatures to generate candidate pairs. The $refine$ step verifies the candidate pairs to get the final results. There are many studies, such as [1], [21] belong to it. The second one such as [15] uses a $trie$ structure to share prefixes and utilizes prefixes for pruning. Vernica et al. propose a parallel approach using MapReduce to solve set-similarity join problem in [14]. It fits for the situation in which the dataset is very large.

Recently, several work has been presented to deal with the spatio-textual similarity join, such as [12], [2]. In [12], several methods have been proposed based on prefix filter technique. In [2], Bouros et al. combine ideas from spatial distance join and set similarity join methods and propose algorithms that take into account both spatial and textual constraints. However, Both of these methods are centralized and run on a single machine.

## VII. Conclusion

In this paper, we study a new problem called spatio-textual $k$ nearest neighbor join (ST-$k$NNJ) in sensor networks. Different from traditional approaches to answer $k$ nearest neighbor ($k$NN) query, we focus on not only spatial information but also text information collected by sensors. ST-$k$NNJ organizes the queries into query set for batch processing. Furthermore, we solve this problem in the clouds by utilizing distributed computing framework MapReduce. We formolize the problem of ST-$k$NNJ and present a naive approach and an improved approach. The former one adopts block nested loop join to do brute-force search on query set and sensor set to get query results. The latter one uses filter-and-refine framework to get text-similar candidates and then to find final $k$NNs. At last, we design and conduct experiments to evaluate the proposed approaches and the experiment results show that the improved approach gains better query performance than the naive approach. In the future work, we desire to design more efficient approaches to make full use of the spatial information to get $k$NNs and conduct extensive experiments to compare the performance between our approaches and the traditional approaches.

## Acknowledgment

This work is partially sponsored by the National Basic Research 973 Program of China (No. 2015CB352403), the National Natural Science Foundation of China (NSFC) (No. 61261160502, No. 61202025), the Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT), the Scientific Innovation Act of STCSM (No. 13511504200), and the EU FP7 CLIMBER project (No. PIRSES-GA-2012-318939).

## References

[1] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[2] P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. *VLDB*, 6(1):1–12, 2012.

[3] Y. Cai, K. A. Hua, and G. Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *MDM*, pages 27–38, 2004.

[4] C.-Y. Chow, M. F. Mokbel, and H. V. Leong. On efficient and scalable support of continuous queries in mobile peer-to-peer environments. *TMC*, 10(10):1473–1487, 2010.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[6] T.-Y. Fu, W.-C. Peng, and W.-C. Lee. Parallelizing itinerary-based knn query processing in wireless sensor networks. *TKDE*, 22(5):711–729, 2010.

[7] B. Gedik and L. Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67–87, 2004.

[8] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD*, pages 479–490, 2005.

[9] Y. Komai, Y. Sasaki, T. Hara, and S. Nishio. Knn query processing methods in mobile ad hoc networks. *TMC*, 13(5):1090–1103, 2014.

[10] F. Li, B. Yao, M. Tang, and M. Hadjieleftheriou. Spatial approximate string search. *TKDE*, 25(6):1394–1409, 2013.

[11] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *VLDB*, 5(3):253–264, 2011.

[12] S. Liu, G. Li, and J. Feng. Star-join: Spatio-textual similarity join. In *CIKM*, pages 1016–1027, 2012.

[13] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *VLDB*, 5(10):1016–1027, 2012.

[14] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506, 2010.

[15] J. Wang, G. Li, and J. Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *VLDB*, 3(1):1219–1230, 2010.

[16] J. Winter and W.-C. Lee. Kpt: A dynamic knn query processing algorithm for location-aware sensor networks. In *DMSN*, pages 119–124, 2004.

[17] J. Winter, Y. Xu, and W.-C. Lee. Energy efficient processing of k nearest neighbor queries in location-aware sensor networks. In *MobiQuitous*, pages 281–292, 2005.

[18] S.-H. Wu, K.-T. Chuang, C.-M. Chen, and M.-S. Chen. Diknn: An itinerary-based knn query processing algorithm for mobile sensor networks. In *ICDE*, pages 456–465, 2007.

[19] S.-H. Wu, K.-T. Chuang, C.-M. Chen, and M.-S. Chen. Toward the optimal itinerary-based knn query processing in mobile sensor network. *TKDE*, 20(12):1655–1668, 2008.

[20] C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: An efficient method for knn join processing. In *VLDB*, pages 756–767, 2004.

[21] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *VLDB*, 1(1):933–944, 2008.

[22] B. Xu, F. Vafaee, and O. Wolfson. In-network query processing in mobile p2p databases. In *GIS*, pages 207–216, 2009.

[23] Y. Xu, T.-Y. Fu, W.-C. Lee, and J. Winter. Processing k nearest neighbor queries in location-aware sensor networks. *SIGPRO*, 87(12):2861–2881, 2007.

[24] B. Yao, F. Li, and P. Kumar. K nearest neighbor queries and knn-joins in large relational databases (almost) for free. In *ICDE*, pages 4–15, 2010.

[25] C. Yu, B. Cui, S. Wang, and J. Su. Efficient index-based knn join processing for high-dimensional data. *INSFOF*, 49(4):332–344, 2007.

[26] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, pages 631–642, 2005.

[27] C. Zhang, F. Li, and J. Jestes. Efficient parallel knn joins for large data in mapreduce. In *EDBT*, pages 38–49, 2010.

[28] Y. Zhang, B. Hull, H. Balakrishnan, and S. Madden. Icedb: Intermittently-connected continuous query processing. In *ICDE*, pages 166–175, 2007.

[29] B. Zheng, J. Xu, W.-C. Lee, and L. Lee. Grid-partition index: A hybrid method for nearest-neighbor queries in wireless location-based services. *VLDB*, 15(1):21–39, 2006.