Scalable and Fast Top-k Most Similar Trajectories Search Using MapReduce In-Memory

Douglas Alves Peixoto(✉) and Nguyen Quoc Viet Hung

The University of Queensland, Brisbane, Australia
{d.alvespeixoto,q.nguyen}@uq.edu.au

Abstract. Top-k most similar trajectories search (k-NN) is frequently used as classification algorithm and recommendation systems in spatial-temporal trajectory databases. However, k-NN trajectories is a complex operation, and a multi-user application should be able to process multiple k-NN trajectories search concurrently in large-scale data in an efficient manner. The k-NN trajectories problem has received plenty of attention, however, state-of-the-art works neither consider in-memory parallel processing of k-NN trajectories nor concurrent queries in distributed environments, or consider parallelization of k-NN search for simpler spatial objects (i.e. 2D points) using MapReduce, but ignore the temporal dimension of spatial-temporal trajectories. In this work we propose a distributed parallel approach for k-NN trajectories search in a multi-user environment using MapReduce in-memory. We propose a space/time data partitioning based on Voronoi diagrams and time pages, named Voronoi Pages, in order to provide both spatial-temporal data organization and process decentralization. In addition, we propose a spatial-temporal index for our partitions to efficiently prune the search space, improve system throughput and scalability. We implemented our solution on top of Spark's RDD data structure, which provides a thread-safe environment for concurrent MapReduce tasks in main-memory. We perform extensive experiments to demonstrate the performance and scalability of our approach.

1 Introduction

GPS trajectory data carry rich information about moving objects, and have been extensively used for a great number of real-world applications, such as city traffic planing, alternative routes suggestion, trip recommendation, drivers pastern analysis, dynamic event identification, and so on [7,15,32].

Given a query trajectory T, a constant k, a time interval $[t_0, t_1]$, and a trajectory dataset S, the top-k nearest neighbor trajectories problem (k-NN), is to find in S the k closest (or most similar) trajectories from T active during $[t_0, t_1]$. k-NN trajectories is one of the most traditional query operations in trajectory databases, and has received plenty of attention, e.g [6,20,23,25]. Applications include, for example, to identify the top-k vehicle's trajectories in a frequent

© Springer International Publishing AG 2016
M.A. Cheema et al. (Eds.): ADC 2016, LNCS 9877, pp. 228–241, 2016.
DOI: 10.1007/978-3-319-46922-5_18

path in order to calculate their average fuel consumption during a certain period of time, for logistics optimization. However, processing k-NN trajectories in a multi-user environment is challenging; the application may be serving hundreds of requests over the network, and k-NN search in general demands extensive use of computational resources. Furthermore, k-NN search for trajectories is a complex operation, unlike other simpler spatial objects, trajectories are essentially non-uniform sequential data with variable length, attached with both spatial and temporal attributes.

Besides, the massive amount of GPS data available, as well as the increasing number of trajectory data application users, demands more robust, fast and scalable solutions, since location-based service should be able to serve multiple requests over large-scale datasets. Therefore, a typical solution is to consider distributed parallel computation with frameworks such as MapReduce (MR) [8], which provides an abstraction for parallel computation and efficient resources allocation of concurrent threads. Frameworks like Spark [29], on the other hand, provides a MR solution for faster data processing using in-memory data storage.

Current state-of-the-art for k-NN trajectories, however, mainly focus on single-thread/single-user paradigm, and cannot be easily tailored to the MR model [5,6,20,23]. Existing research to support spatial queries using MR, e.g. [1,3,10,14], utilize either a multi-core *divide-and-conquer* strategy, where each *mapper* is responsible to process a sub-query over a subset of the dataset, while the intermediate results from the *map* are refined by the *reducers*; or utilize spatially-aware partitioning in order to organize the space into disjoint groups of spatially close objects. Spatial-aware partitioning strategies in MR can achieve up to 10x faster performance than *divide-and-conquer* by maintaining data locality [10,33], since only a smaller number of partitions containing query candidates are selected for processing, reducing query latency and avoiding unnecessary I/O.

Contribution. The current MR works on k-NN, however, either apply for the spatial dimension only, ignoring the sequential nature and temporal dimension of trajectories, i.e. [2,12,14,31]; or only supports range selection for trajectories [16,27]. To overcome this limitation, we propose a bulk-loading in-memory partitioning strategy based on Voronoi diagrams and time pages, named **Voronoi Pages**, to support multiple k-NN trajectories query in MR, and a spatial-temporal composite index, named **VSI** (Voronoi Spatial Index) and **TPI** (Time Page Index), to prune the search space and speed up trajectory similarity search. Voronoi-based partitioning have been successfully used for distance-based search in MR [2,12,14]. For the best of our knowledge, this is the first work to address similarity-based search for trajectory data in MR.

2 Problem Statement and Overall Approach

2.1 Top-k Trajectories Problem

A trajectory T of a moving object is a multidimensional sequence of spatial-temporal points, where each point is described as a triple (x, y, t), where (x, y)

are the spatial coordinates of the moving object at a time t. We need a distance function $d(T_a, T_b)$ to calculate the distance between two trajectories [24]. In this work, we adopt the Edit Distance with Projections (EDwP) [20], as trajectory similarity function. EDwP is threshold-free and can cope with local time shifts and non-uniform sampling rates, which are essential in real-world trajectory datasets.

Problem Statement (*k*-NN). Given an input trajectory dataset $\mathbb{S}$, a trajectory distance function $d(T_a, T_b)$, and a batch of queries $\mathbb{Q} = [(T_1, k_1, t_{1a}, t_{1b}), ..., (T_n, k_n, t_{na}, t_{nb})]$ from application users. For each input query $(T_i, k_i, t_{ia}, t_{ib})$ we want to find in $\mathbb{S}$ the k_i closest trajectories from T_i w.r.t. $d(T_a, T_b)$, and active in $[t_{ia}, t_{ib}]$, named k_i-NN(T_i, t_{ia}, t_{ib}).

Our goal is to improve performance and throughput of k-NN trajectory search using MR in-memory, and allow concurrent queries in multi-user servers. The cost of executing a k-NN query can be measured by the number of input records it has to read and process [3]. Following we describe the challenges of processing k-NN search in multi-user environments over large-scale trajectory datasets.

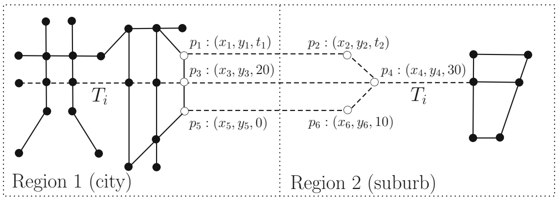


Fig. 1. Road network example.

Temporal dimension: Since trajectory data are queried by both spatial and temporal attributes, temporal dimension must be taken into account [26]; for instance, consider the road network in Fig. 1 connecting two spatial regions, there may be thousands of trajectories passing through the road T_i, however the client may be interested in retrieving similar trajectories within a specific time period. For example, imagine three trajectories passing through $T_1 : \{p_1, p_2\}$, $T_2 : \{p_3, p_4\}$ and $T_3 : \{p_5, p_6\}$, if we want the closest trajectory to T_1 within time $t = [0, 10]$, the application should return T_3 instead of T_2. Only grouping trajectories by spatial region in that case is not strict enough.

Skewness: In Fig. 1 the density of moving object's passing through Region 1 (city region) is much larger than in Region 2 (suburb region). Therefore, we must provide a partitioning strategy as uniform as possible to avoid load imbalance, yet keeping spatial proximity, which is a key factor in spatial data processing in MR [9]. If many concurrent queries are accessing spatial-temporal regions with high density of trajectory points, a poor partitioning may cause I/O bottleneck and impair the system throughput.

System efficiency and reliability: The number of spatial-temporal regions containing the answer of a k-NN query highly depends on the value of k and the time interval $[t_0, t_1]$, hence an iterative neighborhood search may be necessary to select candidate trajectories. In-memory based structures are more suitable for iterative MR jobs, where it is necessary to apply a function repeatedly on the working set of data [29]. Furthermore, we must take concurrency control into account, since different tasks might be working over the same data partitions. Therefore, we provide an off-line data partitioning on top of Spark's in-memory data structure (RDD) [28]. Spark's scheduler is fully thread-safe and supports multiple on-line requests over its RDD.

2.2 Scalable *k*-NN query using Partitioning

An efficient way to answer k-NN trajectories search is to partition the dataset in a spatial-temporal aware manner, such that the amount of data processed by each query is minimized. Spatial-aware partitioning strategies, such as grid cells and Voronoi diagrams (VD), aim to organize the data into smaller partitions of spatially close objects to reduce the number of query candidates, hence reducing network and I/O costs [30,34]. In this work we extend a VD data partitioning for spatial-temporal trajectories in MR, for it maintains data proximity and provides uniform distribution for skewed datasets. VD is particularly suitable for distance-based search, where grid partitioning suffer from a significant loss of pruning power [2,11,12,14].

Overall process. We uniformly partition the space into Voronoi cells using k-Means clustering, and each Voronoi cell into static temporal partitions (i.e. pages). Trajectories are split into sub-trajectories according to their spatial-temporal extent, such that each sub-trajectory is mapped to one Voronoi Page. We build our Voronoi Pages partitions on top of RDDs to speed up query processing. We process a k-NN query in parallel in a *filter-and-refinement* fashion, first filtering candidate pages, and then running a precise check on the candidate pages. Each process unit can manage a number of pages within a RDD in parallel, and concurrent queries can be served by Spark over its RDD.

3 Voronoi-Based Partitioning with Temporal Awareness

Given an input trajectory dataset, we read and split each trajectory into a set of sub-trajectories, according to its spatial and temporal extent, such that each sub-trajectory is assigned to only one spatial-temporal partition.

Space Partitioning. Given a set of n generator pivots in the dataset space, $PV = \{p_1, ..., p_n\}$, where $p_i = (x_i, y_i)$, we partition the dataset space into n disjoint spatial partitions, where each trajectory sample point is assigned to its closest pivot (i.e. Voronoi cell). Figure 2 illustrates eight trajectories, T_1 to T_8, partitioned across seven Voronoi cells, P_1 to P_7. Boundary trajectories, e.g. T_1 and T_4, are split into sub-trajectories, where each sub-trajectory is assigned to its overlapping cell.

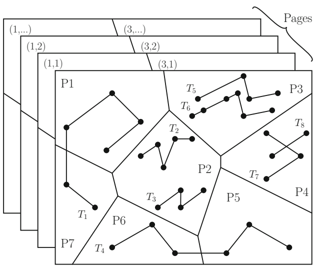


Fig. 2. Trajectories partitioned across Voronoi cells.

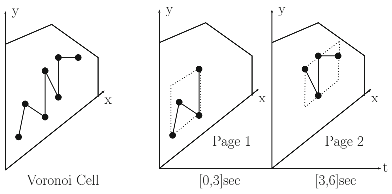


Fig. 3. Sub-trajectory partitioning into Voronoi Pages, $TW = 3\,\text{s}$. Each page contains sub-trajectories that overlap with both the Voronoi polygon area and time window.

Time Partitioning. Given a time window size TW we split the time space of each Voronoi cell into static time pages of size $= TW$, and assign each sub-trajectory sample point inside a polygon to a time page according to its time-stamp. Figure 3 illustrates a sub-trajectory in a given Voronoi cell split into time pages. For the sake of simplicity we assume each sub-trajectory sample point in Fig. 3 was uniformly collected every one second, that is $t_i = [0, 6]\,\text{s}$; however, this approach is for both uniform and non-uniform samples.

Voronoi Page. Each time page for a given Voronoi cell is called a Voronoi Page (VPage), identified by a spatial-temporal index $\langle VSI, TPI \rangle$, where **VSI** (Voronoi Spatial Index) is the page's polygon identifier, and **TPI** (Time Page Index) is the page's time window identifier. Each VPage is composed of two structures: (1) a local R-Tree of sub-trajectories in the page, and (2) a list of the trajectories' IDs in the page. In our implementation, we use simple R-Tree of sub-trajectory bounding boxes. However, any other index access method for sub-trajectories can be used within a VPage.

Handling Boundary Trajectories. While partitioning both space and time we expect some trajectories to intersect more than one VPage. To minimize replication, we split boundary trajectories and replicate only the boundary segments, that is, if a trajectory segment $\overline{p_i p_{i+1}}$ crosses any polygon boundaries, we split the trajectory and assign the boundary segment to both sub-trajectories; each sub-trajectory is assigned to its overlapping polygon. For the temporal dimension the situation is likewise.

Generator Pivots. We choose the number of generator pivots n based on the size of the dataset and the default RDD block size (i.e. 64 MB), so that each task can process data blocks with roughly the same number of polygonal partitions. We study the effect of n for the system performance in Sect. 5.2. In addition, we must choose the pivots in order to break the space into uniform clusters to avoid load imbalance. Therefore, we use the parallel k-Means++ heuristic [4], provided in the Spark machine learning library (MLlib) [17], which provides a fair approximation of the deterministic k-Means.

3.1 MapReduce Implementation

We assume each input file contains one trajectory per line, as a sequence of spatial-temporal points; the data is initially in the HDFS. We build our VPages structure as an RDD with a `map()` and `reduce()` functions as follows. The partitioning process returns an RDD of VPages (i.e. $\mathbb{P}_{RDD}^{pages}$) cached in-memory. Further details can be found in the technical report [21];

Map: The *mapper* reads and splits a trajectory T into m sub-trajectories, according to its spatial-temporal dimension, and emits a list of $\langle (VSI, TPI), T_i^{sub} \rangle$ with m pairs, $i \in [1,..,m]$, consisting of a sub-trajectory T_i^{sub} as *value*, and the spatial-temporal index of the VPage containing T_i^{sub} as *key*.

Reduce: The *reducer* receives a list of sub-trajectories (*values*), and groups them by index (*key*), adding each sub-trajectory to the VPage R-Tree. At the end of the parallel process, the *reduce* returns an RDD of $\langle (VSI, TPI), VPage \rangle$ pairs, consisting of the spatial-temporal VPage index, and the final VPage.

3.2 Trajectory Track Table (TTT)

We must keep track of sub-trajectories across VPages, so that we can retrieve and rebuild a trajectory when processing a k-NN query. For this purpose, we propose a table-like structure, named **Trajectory Track Table** (TTT). The TTT is a in-memory structure, where each tuple of the table is a pair composed of a trajectory ID and a set of references to VPages (page index hash) containing the pages a trajectory intersects with. The TTT is constructed as an RDD (i.e. $\mathbb{T}_{RDD}^{table}$) so that all nodes have access to it without the need of replication. We build the $\mathbb{T}_{RDD}^{table}$ with MR as follows.

Map: The *mapper* reads and map each input trajectory to a list of $\langle T_{id}, (VSI, TPI) \rangle$ pairs, containing the trajectory identifier for each VPage index T_{id} overlaps with.

Reduce: The *reducer* groups indexes by trajectory T_{id} into a set of VPage indexes $\langle T_{id}, \{(VSI, TPI)\} \rangle$. Each pair $\langle T_{id}, \{(VSI, TPI)\} \rangle$ is henceforth called a table tuple.

4 k-NN Trajectories Query Answering

Given a query trajectory Q, and a time interval $[t_0, t_1]$, we want to retrieve the k-NN of Q within the time interval $[t_0, t_1]$. By using a VD-based approach we focus on the spatial proximity to the specified query location. Let $VP(Q)$ be the set of Voronoi polygons covered by Q, and $VP_N(Q)$ be the set of neighbor polygons of $VP(Q)$. To process k-NN trajectory queries we take advantage of the neighborhood properties of VDs as follows. The proof of these properties can be found at [18].

1. The nearest generator pivot p_j from another generator pivot p_i is among the pivots whose Voronoi polygons share edges with $VP(p_i)$ (locality preserving property).
2. Let n and n_e be respectively the number of pivots and the number of edges in a VD, then $n_e \leq 3n - 6$. And given that every edge in a VD is shared by exactly two polygons, then the average number of edges per Voronoi polygon is less equal than six, i.e., $2(3n - 6)/n = 6 - 12/n \leq 6$.

NN Trajectory Search. From property 1, the nearest neighbor NN(Q) of a query object Q is either in $VP(p_i)$, where p_i is the nearest pivot from object Q, or among the neighbor cells of $VP(p_i)$, for Q might be a boundary object. However, because our query object Q is a trajectory, we must check all polygons intersecting with Q and their neighbors. Moreover, we are interested in a spatial-temporal k-NN, thus we have to look in the specific time pages inside each partition. More precisely, assuming our query object is T_4, and we are interested in a time interval $[t_0, t_1]$, we search for the NN(T_4, t_0, t_1) inside the VPages set $\mathbb{F} = \{(2, [t_0, t_1]),\ (4, [t_0, t_1]),\ (5, [t_0, t_1]),\ (6, [t_0, t_1]),\ (7, [t_0, t_1])\}$. Nevertheless, trajectories in $\mathbb{F}$ may span to other VPages depending on their spatial and temporal extent, for instance, T_1 in P_7 also spans to P_1. We must ensure the whole trajectories are returned from the previous step in order to evaluate their distances. Thus, from this point we visit the TTT to retrieve the index of other VPages containing the trajectories in $\mathbb{F}$ (if there is any new). We filter from the $\mathbb{P}_{RDD}^{pages}$ the sub-trajectories in the VPages returned from the $\mathbb{T}_{RDD}^{table}$ and append the remainder sub-trajectories to $\mathbb{F}$. A post-processing step is done to merge sub-trajectories in $\mathbb{F}$, and compute the NN(Q, t_0, t_1).

***k*-NN Trajectories Search.** Similar to [2], to calculate the remainder (k-1)-NN of Q, suppose both Q and NN(Q) are in P_3, $Q = T_5$ and NN$(T_5) = T_6$, thus we also look for the second NN of Q in pages inside the neighborhood of P_3, that is P_1, P_2 and P_4. The remainder NNs are retrieved in the same recursive process; the search stops at the k_{th} iteration if the number of candidates c is $c \geq k$, or continues the search until $c \geq k$. From property 2, the number of neighbor partitions we have to look for time pages in every iteration is at most six for every partition containing the current candidate.

4.1 K-NN Search Using RDD

The VPages containing the k-NN result are unknown until the query is executed, thus, we calculate k-NN(Q, t_0, t_1) with k iterative *filter-and-refinement* MR jobs, so that in every i_{th} iteration we have the i_{th}-NN(Q, t_0, t_1) result. More technical details and implementation can be found at [21].

First Filter-Refinement: We first select from $\mathbb{P}_{RDD}^{pages}$ all pages in the interval $[t_0, t_1]$, for every polygon $P_i \in (VP(Q) \cup VP_N(Q))$. Finally, we collect all trajectories inside the filtered VPages, and active during $[t_0, t_1]$ – we use the $\mathbb{T}_{RDD}^{table}$ to track other VPages containing trajectories in $(VP(Q) \cup VP_N(Q))$. This step returns a RDD of candidate trajectories $\mathbb{T}_{RDD}^{cddt}$. The *refinement* receives

$\mathbb{T}_{RDD}^{cddt}$ from the *filter* step, and returns a list of trajectories from $\mathbb{T}_{RDD}^{cddt}$ sorted by distance to Q. If one is interested in the 1-NN(Q, t_0, t_1) only, we return the first element in the sorted list as the 1-NN(Q, t_0, t_1) result.

Iterative Search: For every i_{th}-NN of Q remaining, we perform a *filter-refinement* process in a fashion as similar as before, using an iterative neighborhood search over the $\mathbb{P}_{RDD}^{pages}$ as stated in Sect. 4. At the end of each i_{th} stage the intermediate results are collected and the candidates list is updated in the application master.

5 Experiments

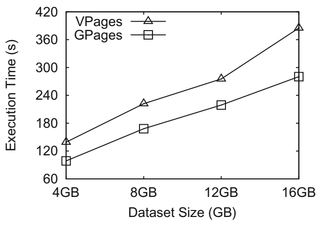
We present a set of highlighted experiments on a real trajectory dataset to evaluate the performance and scalability of our approach.

Experimental Setup. We use a 16 GB trajectory dataset collected from the southern region of China. The dataset contains 4 million heterogeneous trajectories from taxis and personal vehicles in a period of five days. The data is initially stored in HDFS. All algorithms are implemented in the Spark Java library version 1.5.1. Experiments are conducted on a cluster with 30 nodes. Each node is a Ubuntu 14.04 LTS with dual-core processor and 3 GB of RAM. We employ Spark-JobServer [22] to support multiple concurrent jobs in our application. We evaluate our method for both NN and k-NN trajectory queries. Due to space limitations we set $k = 10$ by default. We use $1,000$ as the default number of cells, and fixed the time window at $1,200$ s (based on the mean $\mu = 543$ s and standard deviation $\sigma = 700$ s of trajectories duration), so most trajectories fit into one time page. As query input, we randomly selected 100 trajectories from the dataset, the query time was set as the beginning and ending time of each query trajectory; we perform the queries in batches of 5 concurrent threads by default.

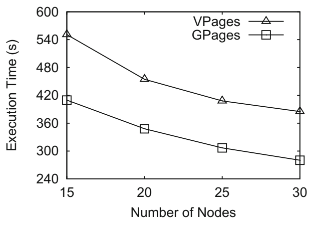
We compare our approach against a Grid-cell based approach, also commonly used in spatial MR works, e.g. [10,27]. The grid-based approach is similar to the VD one, except the space is partitioned into a uniform grid. Throughout this section we refer to the VD approach and Grid approach as *VPages* and *GPages* respectively. To process k-NN queries in *GPages* we employ a technique similar to that in SpatialHadoop [10] to prune the search space, except we use the trajectories' centroid distance to select candidate trajectories. To a fair comparison, we apply the same splitting strategy and spatial-temporal granularity on both implementations.

5.1 VPages Construction Evaluation

Index Construction Scalability. Figure 4(a) demonstrates the execution time for reading the data from HDFS and building both *VPages* and *GPages* RDDs for different dataset sizes, i.e. from 1/4x to 1x the original dataset. *GPages*



(a) By dataset size.



(b) By number of nodes.

Fig. 4. Index construction time evaluation.

Table 1. Trajectories distribution across *VPages* by number of pivots.

#Pivots	#VPages	#Sub-Trajectories	#Splits (Avg)	Latency (s)
250	78,715	6,045,863	1.51	247.5
500	146,479	6,276,712	1.57	301.5
1,000	265,700	6,538,746	1.63	385.0
2,000	464,912	6,949,443	1.74	497.0

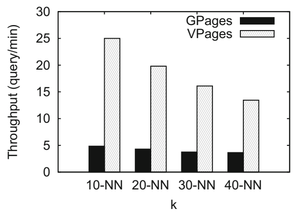


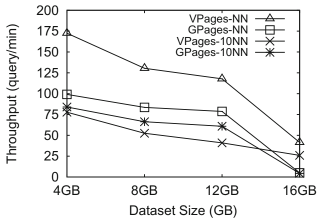
Fig. 5. Query Number of k.

outperformed *VPages* on index construction time on all scenarios due to the one-to-one complexity of parsing trajectory data points to a uniform grid, against the O($n * k$) complexity of Voronoi diagram construction. This is also true for different numbers of computing nodes as shown in Fig. 4(b). However, *VPages* outperformed *GPages* in query latency and throughput as we discuss further on this section.

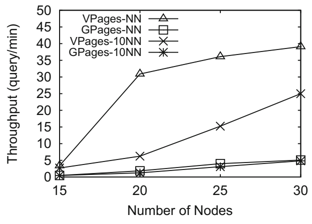
Effect of the Number of Pivots. Table 1 gives statistical information about trajectories distribution across *VPages* and the execution time on building the *VPages* RDD for different numbers of Voronoi cells. As expected, the execution time increase with the number of cells, this is due the increasing number of comparisons during the map phase. The number of trajectories' splits increase with the spatial partitioning granularity, this is due to increasing number of boundary trajectories in more tight partitions. However, system throughput improves for larger numbers of cells as we further discuss.

5.2 System Performance and Scalability

Scalability Evaluation. Figure 6(a) shows the system throughput for NN and 10-NN queries on both *VPages* and *GPages* RDDs. We measure system throughput by the number of queries completed per minute. Overall, *VPages* performed up to 10x better than *GPages* for both NN and k-NN queries as the dataset grows. This is mainly due to two reasons: first the filter step of *VPages* is more accurate than its *GPages* counterpart on filtering candidate trajectories; secondly, *VPages* presented a more uniform data distribution across partition using



(a) By dataset size.

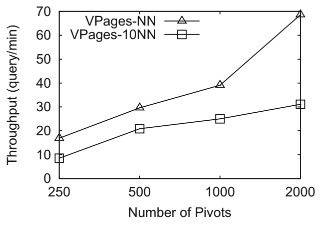


(b) By number of nodes.

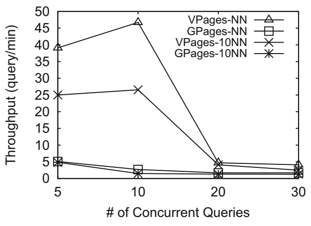
Fig. 6. System throughput evaluation.

k-Means clustering than the grid-based approach, which caused the load imbalance in *GPages*. However, for dataset smaller than 16 GB, *GPages* performed k-NN search slightly better than *VPages*; this is due to the iterative neighborhood search on *VPages*, which seeks for the query result on neighbor cells even for small input datasets. This difference, however, disappears as the dataset grows due to the most homogeneous data distribution of *VPages*. Near 16 GB for *GPages*, however, the cluster resources utilization reaches its limits for the default parameters, once each concurrent query needs to cache and process its own copy of the filtered RDD partitions, which causes Spark to shuffle more data and spill some data to disk for larger input datasets, thus the performance deterioration on *GPages*. The situation is likewise with number of nodes smaller than 20 nodes, as shown in Fig. 4(b), where *VPages* outperformed *GPages* in all scenarios up to 25x in NN search and up to 10x in k-NN.

Effect of the Number of Pivots. Figure 7(a) gives the system throughput using *VPages* for different numbers of Voronoi cells. Overall, finer-grained partitions tends to positively affect query latency and throughput, this is due to the filter step to be more precise when retrieving candidate trajectories. This improvement in query latency increases the resources availability, hence increasing parallelism and system throughput.



(a) By # of pivots.



(b) By # of concurrent queries.

Fig. 7. System throughput by number of pivots and by number of concurrent queries.

Concurrency Evaluation. Here we evaluate the effect of the number of concurrent queries to the system throughput. We submit queries in batches of 5 to 30, and start one thread per query job using the Spark-JobServer [22] framework. Queries are executed in a "round-robin" fashion using Spark's *FAIR* job scheduling, so that all queries get a roughly equal share of cluster resources. Figure 7(b) gives the overall results. For *VPages* on both NN and k-NN queries the system throughput increased from 5 to 10 concurrent queries, this is due to the best use of our cluster resources. Near 10 concurrent queries, however, the resources utilization reaches its peak, hence its maximum throughput. Furthermore, even with dataset in main-memory the overhead of managing large numbers of concurrent jobs can lead to more contentions and strongly limit the system scalability [19]. For *GPages* the situation was much worse, with its peak near 5 concurrent jobs. In summary, *VPages* demonstrated to be up to 15x better than *GPages* on handling multi-user application and concurrent jobs.

Effect of Number of Neighbors (k). Figure 5 gives a comparative on the system throughput as k grows. The partitions containing the k-NN are unknown until the query is executed; however, spatial locality is not always preserved in grid-based, which means we need to extent the search space in *GPages* further than in *VPages* to retrieve the candidate trajectories, which negatively impacts the performance of *GPages*. Due to the locally preserving property of VDs, most neighbors of a given object are in the nearby polygons, thus are retrieved in the first iterations; and due to the homogeneous data distribution in VPages, the number of trajectories in the neighbor partitions to retrieve are roughly the same, which leads to near linear effect on query latency as k increases; the system throughput, therefore, is directly affected by queries latency. *VPages* is more sensitive to k than *GPages*, however, *VPages* is only as poor as *GPages* for very large values of k, where a great number of partitions need to be track.

6 Related Work

MR-based Solutions for k-NN of spatial points. Lu et al. [14] and Akdogan et al. [2] use a VD-based approach to partition the space and index spatial objects based on its closest pivots during the *map* phase, and processing k-NN and RNN queries [2] and k-NN join [14] in iterative *MR* tasks; and outperforms similar MR works based on grid-based partitioning for k-NN query [33] and k-NN join [31]. Our partitioning method is closely related to that in [2,14], except we extend VD for spatial-temporal dimension of trajectories in order to support trajectory similarity search, we also use RDD to support in-memory based computation and concurrent queries.

Unified Frameworks for Spatial Queries in MR. SpatialHadoop [10] has been developed to support spatial data operations and spatial data indexing using MR. SpatialHadoop outperformed Hadoop and traditional approaches by using an extensive set of spatial partitioning structures to improve spatial queries performance in MR [9]. Similarly, Aly et al. proposed AQWA [3], an adaptive spatial data partitioning based on kd-Trees to support range selection and k-NN search in MR; unlike SpatialHadoop, AQWA provides a dynamic space partitioning which reacts to changes on both the query workload and new incoming data. Similar to SpatialHadoop, Hadoop-GIS [1] and ScalaGiST [13] presented another general purpose solution for spatial queries and cost-efficient spatial data indexing in MR. However, the aforementioned works only support nearest neighbors search for spatial points, and do not provide any support for spatial-temporal trajectories.

Centrally-based Indexing for Similarity Search. This includes LCSS [23], ERP [5], EDR [6] and TrajTree [20]. The goal is to extend tree-based indexing to organize the trajectory dataset space for similarity search. However, all these methods index trajectories based on spatial similarity only, ignoring its temporal dimension. The main drawbacks of these centrally-based structures are that they do not provide fully decentralization for parallel computation, and do not scale for large datasets. Furthermore, all aforementioned works are for disk-based computation, whereas our solution takes advantage of in-memory structure to speed up similarity search. SharkDB [25] is a in-memory storage architecture for trajectories, which partitions the dataset into time frames, in order to support general purpose operations. However, SharkDB uses only temporal partitioning, and focus on column-oriented architectures. By using RDD, however, we can provide a distributed and fault-tolerant solution for large datasets and concurrent tasks.

7 Conclusions

In this work we present a multi-user system to process concurrent k-NN trajectories search using Spark's RDD, a thread-safe and resilient distributed data structured for large-scale data processing in main-memory using MapReduce model. We introduced a novel spatial-temporal data partitioning approach, named Voronoi Pages, built on top of RDD to a scalable and fast processing of concurrent k-NN trajectories search in MR. Voronoi Pages provides both homogeneous data partitioning and spatial-temporal locality preserving, essentials for MR-based systems. Our experimental results based on a real trajectory dataset demonstrates the performance and good scalability of our approach against another common approach used in MR for spatial data.

Acknowledgments. This research is partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq).

References

1. Aji, A., et al.: Hadoop-GIS: a high performance spatial data warehousing system over mapreduce. VLDB **6**(11), 1009–1020 (2013)
2. Akdogan, A.: Voronoi-based geospatial query processing with mapreduce. In Cloud- Com, pp. 9–16. IEEE (2010)

3. Aly, A.M., et al.: AQWA: adaptive query workload aware partitioning of big spatial data. VLDB **8**(13), 2062–2073 (2015)
4. Bahmani, B., et al.: Scalable k-means++. VLDB **5**(7), 622–633 (2012)
5. Chen, L., Ng, R.: On the marriage of lp-norms, edit distance. In: VLDB, pp. 792–803 (2004)
6. Chen, L., Özsu, M.T., Oria, V.: Robust, fast similarity search for moving object trajectories. In: SIGMOD, pp. 491–502 (2005)
7. Dai, J., et al.: Personalized route recommendation using big trajectory data. In: ICDE, pp. 543–554 (2015)
8. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
9. Eldawy, A., Alarabi, L., Mokbel, M.F.: Spatial partitioning techniques in SpatialHadoop. VLDB **8**(12), 1602–1605 (2015)
10. Eldawy, A., Mokbel, M.F.: SpatialHadoop: a MapReduce framework for spatial data. In: ICDE, pp. 1352–1363 (2015)
11. Kolahdouzan, M., Shahabi, C.: Voronoi-based k nearest neighbor search for spatial network databases. In: VLDB, pp. 840–851 (2004)
12. Li, C., et al.: Processing moving k NN queries using influential neighbor sets. VLDB **8**(2), 113–124 (2014)
13. Lu, P., et al.: ScalaGiST: scalable generalized search trees for mapreduce systems [innovative systems paper]. VLDB **7**(14), 1797–1808 (2014)
14. Lu, W., et al.: Effcient processing of k nearest neighbor joins using mapreduce. VLDB **5**(10), 1016–1027 (2012)
15. Luo, W., et al.: Finding time period-based most frequent path in big trajectory data. In: SIGMOD, pp. 713–724 (2013)
16. Ma, Q., et al.: Query processing of massive trajectory data based on mapreduce. In: International Workshop on Cloud Data Management, pp. 9–16. ACM (2009)
17. MLlib: http://spark.apache.org/docs/latest/mllib-guide.html
18. Okabe, A., et al.: Spatial tessellations: concepts and applications of Voronoi diagrams, vol. 501. Wiley, New York (2009)
19. Pandis, I., et al.: Data-oriented transaction execution. Proc. VLDB Endowment **3**(1–2), 928–939 (2010)
20. Ranu, S., et al.: Indexing, matching trajectories under inconsistent sampling rates. In: ICDE, pp. 999–1010 (2015)
21. Scalable and fast top-k most similar trajectories search using MapReduce in-memory. Technical report (2016). https://www.researchgate.net/publication/303487238
22. Spark-JobServer: https://github.com/spark-jobserver/spark-jobserver
23. Vlachos, M., Gunopulos, D., Kollios, G.: Discovering similar multidimensional trajectories. In: Agrawal, R., Dittrich, K.R. (eds.) ICDE, pp. 673–684 (2002)
24. Wang, H., et al.: An effectiveness study on trajectory similarity measures. In: ADC, pp. 13–22 (2013)
25. Wang, H., et al.: SharkDB: an in-memory column-oriented trajectory storage. In: CIKM, pp. 1409–1418 (2014)
26. Wang, X., Zhou, X., Lu, S.: Spatiotemporal data modelling, management: a survey. In: TOOLS-Asia, pp. 202–211. IEEE (2000)
27. Yang, B., Ma, Q., Qian, W., Zhou, A.: TRUSTER: TRajectory data processing on ClUSTERs. In: Zhou, X., Yokota, H., Deng, K., Liu, Q. (eds.) DASFAA 2009. LNCS, vol. 5463, pp. 768–771. Springer, Heidelberg (2009)

28. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: USENIX Conference on Networked System Design and Implementation, p. 2 (2012)
29. Zaharia, M., et al.: Spark: cluster computing with working sets. In: USENIX Conference on Hot Topics in Cloud Computing, p. 10 (2010)
30. Zamanian, E., Binnig, C., Salama, A.: Locality-aware partitioning in parallel database systems. In: SIGMOD, pp. 17–30 (2015)
31. Zhang, C., Li, F., Jestes, J.: Effcient parallel kNN joins for large data in MapReduce. In: EDBT, pp. 38–49 (2012)
32. Zheng, Y., Zhou, X.: Computing with Spatial Trajectories. Springer, New York (2011)
33. Zhong, Y., et al.: Towards parallel spatial query processing for big spatial data. In: IPDPSW, pp. 2085–2094. IEEE (2012)
34. Zhou, X., Abel, D.J., Truffet, D.: Data partitioning for parallel spatial join processing. In: Scholl, M., Voisard, A. (eds.) SSD 1997. LNCS, vol. 1262, pp. 178–196. Springer, Heidelberg (1997). doi:10.1007/3-540-63238-7_30