





Article

An Efficient Indexing Approach for Continuous Spatial Approximate Keyword Queries over Geo-Textual Streaming Data

Ze Deng ^{1,2} , Meng Wang ^{1,2}, Lizhe Wang ^{1,2,*} , Xiaohui Huang ^{1,2} , Wei Han ^{1,2} ,
Junde Chu ^{1,2} and Albert Y. Zomaya ³

¹ School of Computer Science, China University of Geosciences, Wuhan 430074, China; deng_ze@163.com (Z.D.); wangmeng_1998@126.com (M.W.); xhhuang@cug.edu.cn (X.H.); weihan@cug.edu.cn (W.H.); chujunde@cug.edu.cn (J.C.)

² Hubei Key Laboratory of Intelligent Geo-Information Processing, China University of Geosciences, Wuhan 430074, China

³ School of Information Technologies, The University of Sydney, Sydney, NSW 2006, Australia; albert.zomaya@sydney.edu.au

* Correspondence: lizhe.wang@gmail.com

Received: 17 November 2018; Accepted: 24 January 2019; Published: 28 January 2019



Abstract: Current social-network-based and location-based-service applications need to handle continuous spatial approximate keyword queries over geo-textual streaming data of high density. The continuous query is a well-known expensive operation. The optimization of continuous query processing is still an open issue. For geo-textual streaming data, the performance issue is more serious since both location information and textual description need to be matched for each incoming streaming data tuple. The state-of-the-art continuous spatial-keyword query indexing approaches generally lack both support for approximate keyword matching and high-performance processing for geo-textual streaming data. Aiming to tackle this problem, this paper first proposes an indexing approach for efficient supporting of continuous spatial approximate keyword queries by integrating *min-wise* signatures into an AP-tree, namely AP-tree⁺. AP-tree⁺ utilizes the one-permutation *min-wise* hashing method to achieve a much lower signature maintenance costs compared with the traditional *min-wise* hashing method because it only employs one hashing function instead of dozens. Towards providing a more efficient indexing approach, this paper has explored the feasibility of parallelizing AP-tree⁺ by employing a Graphic Processing Unit (GPU). We mapped the AP-tree⁺ data structure into the GPU's memory with a variety of one-dimensional arrays to form the GPU-aided AP-tree⁺. Furthermore, a *min-wise* parallel hashing algorithm with a scheme of data parallel and a GPU-CPU data communication method based on a four-stage pipeline way have been used to optimize the performance of the GPU-aided AP-tree⁺. The experimental results indicate that (1) AP-tree⁺ can reduce the space cost by about 11% compared with MHR-tree, (2) AP-tree⁺ can hold a comparable recall and 5.64× query performance gain compared with MHR-tree while saving 41.66% maintenance cost on average, (3) the GPU-aided AP-tree⁺ can attain an average speedup of 5.76× compared to AP-tree⁺, and (4) the GPU-CPU data communication scheme can further improve the query performance of the GPU-aided AP-tree⁺ by 39.4%.

Keywords: continuous query; spatial approximate keyword matching; indexing methods; GPU

1. Introduction

Traditional Geographic Information Systems (GIS) are well-adapted for offline algorithms over static data. In an offline environment, a GIS application is expected to have complete information about

the input static data to be processed [1]. With the proliferation of Global Navigation Satellite System (GNSS)-equipped devices and wireless sensor networks, current GIS applications (e.g., location-based services and webGIS) need to be much more suited to online processing to deal with large volumes of highly dynamic geo-streaming data. A number of successful attempts have been made for the challenge that current GIS applications face. For example, Galić et al. [2] presents a formal framework consisting of data types and operations needed to support geo-streaming data. In [3], a spatio-temporal query language is proposed to process semantic geo-streaming data. Furthermore, Moby Dick [4,5], which is a distributed framework for GeoStreams, has been developed towards efficient real-time managing and monitoring of mobile objects through distributed geo-streaming data processing on large clusters. A more comprehensive introduction about processing GeoStreams is available in [6].

However, the above-mentioned literature only focuses on the spatial dimension of GeoStreams. In fact, a GeoStream holds the textual property as well. Concretely, more recently massive amounts of geo-textual data are generated including geo-tagged micro-blogs, photos with both tags and geo-locations, points of interests (POIs) and so on [7,8]. For example, according to the descriptions in [9], about 30 million users send geo-tagged data into the Twitter services, and 2.2% of the global tweets (about 4.4 million tweets a day) provide location data together with the text of their posts. These data often come in a rapid streaming fashion in many important applications such as social networks (e.g., Facebook, Flickr, FourSquare and Twitter) and location-based services (e.g., location-based advertising) [9]. Monitoring the geo-textual streaming data is critical to efficiently support above-mentioned GIS applications. For instance, in tweeter applications, users often subscribe some requests containing both location information and textual content. Thus, the GIS applications need to monitor incoming geo-textual streaming data to discover matched messages and notify the users during a period of time [7]. The continuous queries are an effective technique for monitoring purposes over streaming data. These queries are issued once and then logically are executed continuously over data streams to provide a prolonged perspective on the changes of streaming data [10]. However, periodic query re-execution is a well-known expensive operation [11]. The optimization of continuous query processing is still an open issue in the community of data stream management in the past two decades. For geo-textual streaming data, the performance issue of continuous query is more serious since both location information and textual description need to be matched for each incoming streaming data tuple.

The query indexing approaches are dominant for the optimization of continuous query processing, since they can avoid expensive operations of index maintenances comparing to the data indexing alternatives [12]. Recently, some query indexing methods have been proposed to address this performance issue. The systems can quickly filter stream data tuples by index structure built over spatial-keyword queries. These indexing methods have roughly been categorized into three classes: the keyword-first indexing method (e.g., Ranked-key Inverted Quadtree (RQ-tree) [9]), the spatial-first indexing method (e.g., R^+ -tree [13] and Inverted File Quadtree (IQ-tree) [7]), and the adaptive indexing method based on location information and textual information (e.g., Adaptive spatial-textual Partition tree (AP-tree) [9]). However, existing continuous spatial-keyword query indexing approaches are faced with two drawbacks for current applications. First, these existing indexing structures lack the support of approximate keyword matching. The approximate keyword matching is necessary when users have a fuzzy search condition, or a spelling error when submitting the query, or the strings in the database contain some degree of uncertainty or errors. The keyword search for retrieving approximate string matches often is required when searching geo-textual objects, according to the descriptions in [14]. Meanwhile, because geo-textual streaming data tuples arrive rapidly from data sources, high-performance data processing is a key requirement for current continuous query methods. Therefore, there is a need for an efficient continuous query indexing approach for spatial approximate keyword query over geo-textual streaming data.

To address these research challenges, we first employ an approximate string search method to enhance the AP-tree indexing structure to support approximate keyword matching. Furthermore, a GPU platform is used to improve the query performance of our query indexing method.

The main contributions of this study are as follows:

1. We have introduced an advanced AP-tree indexing method called AP-tree⁺ to support continuous spatial approximate keyword queries with efficiently embedding *minimum-wise* (*min-wise*) signatures into the AP-tree structure based on one-permutation *min-wise* hashing [15].
2. We designed a parallel version of AP-tree⁺ on a GPU platform, which further improved the performance of our indexing structure for fast processing geo-textual streaming data. The GPU-aided method parallelized the approximate keyword matching of AP-tree⁺ based on a *min-wise* parallel hashing algorithm [16].
3. We further employed a data streaming communication method [17] to optimize I/O overheads between GPU and CPU during continuously processing geo-textual streaming data.

Additionally, this study only handled the range query amongst a variety of continuous queries since our study is based on AP-tree and the spatial query related to AP-tree is range query.

This paper has been expanded from its previous conference version [18] to include one description about a parallel scheme of approximate keyword matching, a data streaming communication method between CPU and GPU in [17] that is used to optimize the performance of GPU-aided AP-tree⁺, an additional data set for experiments, and more experiments for evaluating proposed indexing methods. The remainder of this paper is organized as follows: Section 2 discusses work relating to indexing methods for geo-textual data. Section 3 presents our materials and methods. Section 4 presents the experimental results and discussions. Section 5 concludes with a summary.

2. Related Work

This section describes the most salient works along indexing methods for geo-textual data. The indexing methods for geo-textual data can be roughly classified into two categories, i.e., for static data and streaming data. The approaches for static data consider that all spatial objects have been stored in a spatial database and each spatial object is described with a set of keywords. Thus, these methods indexed both the location information and textual keywords of each spatial object to support spatial-keyword queries. Among these methods, the R-tree has been widely extended to support geo-textual data. For instance, authors in [19] proposed a hybrid indexing structure that maintains classical inverted lists for rare document terms and additional extended R-trees for more frequent geo-textual terms. Similarly, Zhang et al. proposed an Information Retrieval R-tree (IR-tree) [20] that is the combination of R-tree and inverted files for searching geo-textual data. Some other spatial indexing structures have also been used for geo-textual data. For example, an inverted linear Quadtree (IL-Quadtree) [21] based on the linear Quadtree and inverted index was presented to deal with the problem of top-k spatial keyword search. In [22] an inverted-KD tree was developed for indexing geo-textual data.

With the emergences of social networks and location-based services, these geo-textual data often come in a rapid streaming fashion. The indexing solutions for static geo-textual data cannot directly apply to the geo-textual streaming data. As a consequence, recently a few query indexing attempts, which index continuous queries to filter geo-textual streaming data, have been devoted to address this issue. These indexing methods have roughly been categorized into three classes: the keyword-first indexing methods, the spatial-first indexing ones and the adaptive indexing ones. The representative of keyword-first indexing method is RQ-tree [9]. The RQ-tree first uses ranked-key inverted list that stores least frequent keywords to partition queries into the posting lists. Then multiple Quadtrees are built based on each posting list. On the contrary, the spatial-first indexing methods prioritized the spatial factor for the index construction regardless of the keyword distribution of the query set. For example, IQ-tree [7] employs a Quadtree to organize queries so that each query is attached to one or multiple Quadtree cells. Every query in each cell is assigned to the posting list of its frequent

keyword by a ranked-key inverted list. Similarly, R^t -tree [13] first uses a R-tree to index queries based on their search regions and then each R-tree node records the keywords of its descendant queries for textual filtering purpose. The tree structures of both IQ-Tree [7] and R^t -tree [13] are only determined by the spatial feature. Thus their overall performances are unavoidably deteriorated for different keyword and location distributions of the query workload. So, Zhang et al. proposed an adaptive spatial-textual partition tree (AP-tree [9]) that uses f -ary tree structure so that the queries are indexed in an adaptive and flexible way with respect to the query workload. However, current indexing methods for continuous spatial-keyword queries lack the supports for approximate keyword search.

In [14], authors proposed a Min-wise signature with linear Hashing R-tree (MHR-tree) that is the one by combining R-tree and *min-wise* signatures to deal with spatial approximate keyword queries. Unlike MHR-tree, our indexing structure is towards continuous spatial approximate keyword queries over streaming data while the MHR-tree is for one-pass queries over static data. And then, the MHR-tree belongs to the spatial-first indexing scheme while our method is the adaptive scheme according to query workload. Furthermore, the MHR-tree is based on a family of ℓ *min-wise* independent permutations, which may incur high maintenance costs in the case of dynamic continuous queries. Compared with the MHR-tree, our indexing approach can overcome this issue by one-permutation hashing method in [15] to generate signatures instead of ℓ *min-wise* independent permutations. Finally, our indexing approach considers the GPU platform.

In contrast to the existing indexing methods for the continuous spatial-keyword queries, this paper focuses on the challenges of (1) approximate search of keywords and (2) enabling a high-performance solution to maintain the computational performance of the proposed indexing approach for streaming data. The proposed method is the first indexing method for parallel processing continuous spatial approximate keyword queries.

3. Materials and Methods

In this section, firstly our problem is formulated, and the background knowledge about AP-tree is provided, and then an advanced AP-tree called AP-tree⁺ is proposed. Finally, the GPU-aided AP-tree⁺ is presented.

3.1. Problem Formulation

In the following content, firstly the notations of a geo-textual data stream are provided. Then, the definitions about the continuous spatial approximate keyword query over the geo-textual data stream are introduced. Finally, the problem is stated.

Definition 1 (Geo-textual tuple). A geo-textual tuple, denoted as $t = (\phi, loc, ts)$, is a textual message with geo-location, where ϕ is a set of distinct keywords from a vocabulary set, loc is a geo-location, and ts is the timestamp to label the creation time of the tuple.

Definition 2 (Geo-textual data stream). A geo-textual data stream, denoted as $S = \{t_i | i \in [1, +\infty) \wedge t_i.ts \leq t_{i+1}.ts\}$, is an unbounded data set of geo-textual tuples in timestamp order.

Definition 3 (Continuous spatial approximate keyword query). A continuous spatial approximate keyword query, defined as $q = (\psi, r)$ where ψ is a set of distinct keywords, and r is a range region, is a long-running query until it is deregistered. A geo-textual tuple t in S matches q if and only if the following two conditions are satisfied: (1) the similarity between $t.\phi$ and $q.\psi$ is enough (i.e., $sim(t.\phi, q.\psi) \geq \tau$ where τ is a similarity threshold $\in [0, 1]$), and (2) the $t.loc$ is within the $q.r$.

In this paper, given a set Q of continuous spatial approximate keyword queries, for each incoming tuple t from a geo-textual data stream S , we aim to employ indexing technique over Q to rapidly deliver t to approximate matching queries. Abbreviations summarizes the mathematical notations.

3.2. AP-Tree

Adaptive spatial-textual Partition tree (AP-tree for short) is a f -ary tree where queries are recursively divided by spatial or keyword partitions (nodes).

Given a set of spatial-keyword queries Q , an AP-tree is constructed by employing *keyword partition* and *spatial partition* methods to recursively divide Q in a top-down manner. Under the assumption that there is a total order among keywords in the vocabulary, the *keyword partition* method assigns queries to a node N that is called as a *keyword node* and then partitions the queries into f ordered cuts according to their N_l -th keywords, where N_l is the partition offset of the node N . An ordered cut is an interval of the ordered keywords, denoted as $[w_i, w_j]$, where w_i and w_j ($w_i \leq w_j$) are boundary keywords and $[w_i, w_i]$ is denoted as $[w_i]$ if there is only one keyword in the cut. The *spatial partition* method recursively partitions the space region of a node N called as a *spatial node* into f grid cells and pushes each query into corresponding grid cells whose space regions overlap the region of query. One leaf node of AP-tree is called as a *query node* that holds at most θ_q queries.

As the AP-tree structure is constructed in an adaptive way to query workload by carefully choosing keyword or spatial partitions, two partition methods are measured by a cost model shown in Formula (1).

$$C(P) = \sum_{i=1}^f w(B_i) \times p(B_i) \quad (1)$$

where, P is a partition over a set of queries on one node. $C(P)$ is the expected matching cost about the partition P . Meanwhile B is a bucket of the partition, $w(B)$ is the B 's weight which is the number of queries associated to B , and $p(B)$ is the probability that B is explored during the query matching. Given the object workload can be simulated by query workload, $p(B)$ can be introduced as the following equation:

$$p(B) = \begin{cases} \sum_{w \in B} p(w) & \text{if node is keyword node} \\ \frac{Area(B)}{Area(N)} & \text{if node is spatial node} \\ \text{No partition} & \text{else} \end{cases} \quad (2)$$

In Equation (2), $p(w) = \frac{freq(w)}{\sum_{w \in P} freq(w)}$ where $freq(w)$ is the frequency of keyword w among all queries in Q . $Area(B)$ is the area of the bucket (i.e., cell) B and $Area(N)$ is the region size of the node N . The optimal keyword partition of and spatial partition can be achieved by a dynamic programming algorithm and a local improvement heuristic algorithm introduced in [9], respectively.

In addition, for each keyword node N , a query q is assigned to a dummy cut if N cannot find a cut as there is no enough query keywords, i.e., $|q.\psi| < N_l$. Similarly, each spatial node N has a dummy cell for queries whose region contain the region of N .

An example is given for illustrating the structure of AP-tree in Figure 1. As we can see, given a set of spatial-keyword queries Q shown in Figure 1a, an AP-tree (shown in Figure 1b) is constructed by employing keyword partition and spatial partition methods to recursively divide Q in a top-down manner. In our example, the spatial partition first is chosen to generate one *spatial node* S_1 -node with four cells as the spatial partition is more beneficial to pruning data objects than keyword partition based on the query workload in Figure 1a. Then, according to the queries assigned to C_1 in S_1 -node (i.e., q_3, q_4, q_5, q_6 and q_{10}), one keyword node k_1 -node with three cuts (i.e., $[W_1]$, $[W_2, W_3]$, and $[W_4]$) is created by keyword partition. Meanwhile, a spatial node S_2 -node with four cells is generated based on the queries in C_2 (i.e., q_1 and q_8). In addition, the cell C_3 and C_4 link to two *query nodes*, respectively.

Furthermore, for the *keyword node* k_1 -node, $[W_1]$ links to a *query node* contains q_5 and $[W_4]$ points to another new *query node* holding q_4 . According to the workload of queries (i.e., q_3, q_6 , and q_{10}) in cut $[W_2, W_3]$, a new *keyword node* k_2 -node with two cuts (i.e., $[W_5]$ and $[W_6]$) is created. The k_2 -node points to two *query nodes* (i.e., one *query node* contains q_6 , and the other holds q_3 and q_{10}). For the *spatial node* S_2 -node, the cell C_2 and C_4 also link to two *query nodes*, individually.

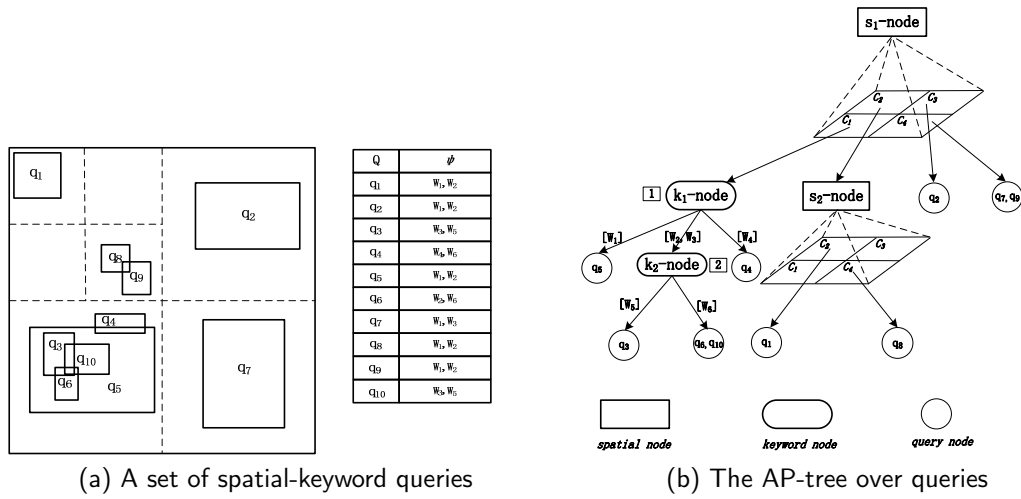


Figure 1. An example of Adaptive spatial-textual Partition (AP)-tree.

3.3. AP-Tree⁺

In this section, we introduce an advanced AP-tree called AP-tree⁺ to support the approximate keyword matching between queries and streaming data tuples.

3.3.1. Developing AP-Tree⁺

The key issue in achieving approximate keyword matching is to define the similarity between the set of query keywords and textual string of a streaming data tuple. The edit distance based on q-grams has been widely applied for approximate string matching (e.g., in [23–26]). The main idea behind these methods is to utilize q-grams as signatures to gain strings similar to a query only if they share common signatures (i.e., q-grams) with the query.

In this paper, we attempt to incorporate q-grams into AP-tree to support approximate keyword matching as well. We call the AP-tree embedding q-grams as AP-tree⁺. Since the textual message of one query q in one query node is a set of distinct and ordered keywords represented as $\{W_1, W_2, \dots, W_m\}$, one straightforward approach is to embed q-grams of all keywords in q represented as $\{GW_1, GW_2, \dots, GW_m\}$ into AP-tree. In fact, for space saving, we do not have to store q-grams of all keywords of q in a query node, due to the fact some keywords of one query have been indexed in keyword nodes of AP-tree. To explain this issue, we first present a Lemma:

Lemma 1. Given a query q with a set of ordered keywords $\{W_1, W_2, \dots, W_m\}$ in one query node q-node, the first n keywords (where $0 \leq n \leq m$) in $\{W_1, W_2, \dots, W_m\}$ can be found in keyword nodes if there are n keyword nodes in q-node's parent node and ancestor nodes.

Proof of Lemma 1. According to the construction of AP-tree (see Section 3.2. AP-tree), the query q may at most be indexed by m keyword nodes. The indexing rule is as following: the first keyword W_1 of q is indexed in one of cuts in one keyword node if the keyword node exists, and then the following keyword node stores the second keyword W_2 of q if the keyword node exists and so on. At most, all keywords can be stored in keyword nodes. For instance, in Figure 1, for a query q_5 with keywords $\{W_1, W_2\}$, the first keyword W_1 has been indexed in k_1 -node, and for another query q_3 with keywords $\{W_2, W_6\}$, W_2 and W_6 have been respectively indexed in k_1 -node and k_2 -node. Meanwhile, in the worst case, there is no one keyword node that indexes any keyword of q . For example, for the query node storing q_1 , both its parent node (i.e., S_2 -node) and ancestor node (i.e., S_1 -node) are spatial node. \square

Based on Lemma 1, we embed q-grams into one AP-tree using the following rules:

1. For one *query node* q-node, if there is no one belongs to *keyword node* in q-node's parent node and ancestor nodes belong to keyword node, we store q-grams of its all keywords for each query in q-node.
2. For one *query node* q-node, if there are n nodes belongs to *keyword node* in q-node's parent node and ancestor nodes, we hold q-grams of the last $m-n$ keywords for each query with m keywords in q-node.
3. We store q-grams of keywords in each cut in every *keyword node*.

3.3.2. Improving AP-Tree⁺

As can be observed, AP-tree⁺ can effectively support approximate keyword matching by embedding q-grams into *query nodes* and *keyword nodes*. However, according to the description in [14], the problem with q-grams is that it may introduce high storage overhead and increase the query cost for large sets of keywords. In our setting, this problem also exists when most indexing nodes in AP-tree⁺ are keyword nodes and there are many cuts in most *keyword nodes*. To address this issue, we first employ a *min-wise* signature-based [27] method in [14] to reduce the storage cost caused by holding large amount of q-grams. Then, we introduce how to accelerate the procedure of generating the large number of *min-wise* signatures.

Reducing Storage Cost with Min-Wise Signatures

According to [14], given a family of *min-wise* independent permutations F , for a set X and any element $x \in X$, when π is chosen at random from F , the following equation holds:

$$\Pr(\min\{\pi(X)\} = \pi(x)) = \frac{1}{|X|} \quad (3)$$

where, $\pi(X)$ is a permutation of X and $\pi(x)$ is the location value of x in the resulted permutation, and $\min\{\pi(X)\} = \min\{\pi(x) | x \in X\}$. With ℓ *min-wise* independent permutations from F , the *min-wise* signature of X is defined as:

$$S(X) = \{\min\{\pi_1(X)\}, \min\{\pi_2(X)\}, \dots, \min\{\pi_\ell(X)\}\} \quad (4)$$

Thus, the set resemblance of two sets A and B defined as $\rho(A, B)$ can be estimated by the similarity of their *min-wise* signatures $S(A)$ and $S(B)$ defined as $\hat{\rho}(S(A), S(B))$. That means $\rho(A, B)$ can be computed by the following equation:

$$\rho(A, B) = \hat{\rho}(S(A), S(B)) = \frac{|\{i | \min\{\pi_i(A)\} = \min\{\pi_i(B)\}\}|}{\ell} \quad (5)$$

Noted that, since the actual permutation constitutes an expensive operation, a two-universal (2U) hash function in [28] is used to simulate such permutations.

In our setting, the implementation of ℓ *min-wise* independent permutations is as following:

1. gain a set of all keywords SK from the query workload.
2. extract a universe set of ordered q-grams U with D dimensions from SK .
3. randomly generate ℓ permutations $\{\pi_1, \dots, \pi_\ell\}$ from U .

Additionally, for k sets A_1, \dots, A_k , the *min-wise* signature of union of A_1, \dots, A_k can be computed by combining the *min-wise* signature of individual sets (see, Equation (6)).

$$S(A_1 \cup \dots \cup A_k)[i] = \min\{S(A_1)[i], \dots, S(A_k)[i]\} \quad (6)$$

Using Equation (4), we can only store the *min-wise* signature of q-grams of each keyword instead of q-grams to reduce the storage cost of AP-tree⁺. However, for the case that there are multiple keywords in one cut of one keyword node, there still exist multiple *min-wise* signatures. We can further reduce the space cost by merging multiple *min-wise* signatures into one *min-wise* signature

based on Equation (6). Let G_W be a q-gram set of keyword W , the scheme is illustrated in Figure 2. We store *min-wise* signatures of q-grams of keywords in *query nodes* and *keyword nodes*. For example, for one *query node* holding q_5 with a keyword set W_1, W_2 , we only store S_1 that represents the *min-wise* signature of G_{W_2} since W_1 has been indexed in k_1 -node. Similarly, for another *query node* holding q_1 with a keyword set W_1, W_2 , due to the reason there no any keyword node that hold W_1 and W_2 among its parent node and ancestor nodes, we store *min-wise* signatures of both G_{W_1} and G_{W_2} (i.e., S_7 and S_8) into this *query node*. On the other hand, for *keyword nodes*, we directly store the *min-wise* signature of q-grams of the keyword if there is only one keyword in one cut, while the *min-wise* signature of union of q-gram sets of these keywords is held if there are more than one keyword in one cut. For instance, k_2 -node has two cuts (i.e., $[W_5], [W_6]$), thus we input the *min-wise* signature of G_{W_5} (i.e., S_2) and W_6 (i.e., S_3) into k_2 -node. For k_1 -node with three cuts (i.e., $[W_1], [W_2, W_3], [W_4]$), we input three *min-wise* signatures (i.e., S_4, S_5, S_6). Among three signatures, S_4 is the *min-wise* signature of G_{W_1} , S_5 is the one of $G_{W_2} \cup G_{W_3}$, and S_6 is the *min-wise* signature of G_{W_4} .

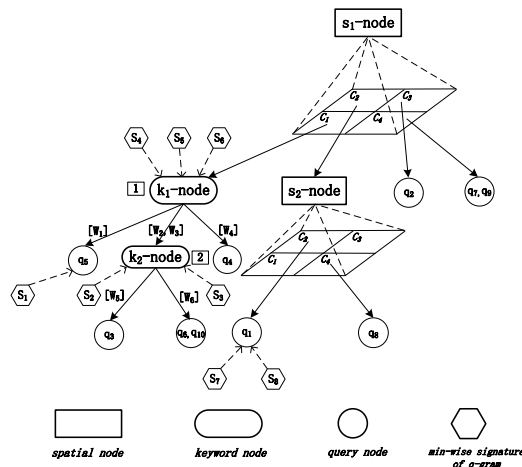


Figure 2. An example of AP-tree⁺.

Accelerating The Generation of *Min-Wise* Signatures

In the aforementioned method, we employ a family of ℓ *min-wise* independent permutations F to create *min-wise* signatures of q-grams. However, the major drawback of this *min-wise* hashing method is the expensive preprocessing cost, as the method requires applying a large number of permutations on the data [29]. For example, in [14] $\ell = 50$ permutations have been used for constructing a MHR-tree. According to our previous works, the continuous queries hold dynamic property [30], thus AP-tree⁺ may suffer from frequent updates, as a result, the *min-wise* signatures need to be frequently recomputed as well. To address this problem, we used an one-permutation *min-wise* hashing method in [15] to generate signatures instead of ℓ permutations. The one-permutation method breaks the space evenly into ℓ bins, and stores the smallest nonzero in each bin, instead of only storing the smallest nonzero in each permutation and repeating the permutation ℓ times. For example in Figure 3, consider two sets of q-grams $S_1, S_2 \subseteq U$ with $D = 12$, a sequence of index of one permutation π from U is defined as $I = \{0, 1, \dots, 11\}$, V_1, V_2 are two binary (0/1) data vectors for representing locations of the nonzeros in π . We equally divide the sequence I into three bins and find the smallest nonzero element in each bin to generate $\pi(V_1) = [0, 5, 8]$ and $\pi(V_2) = [1, 6, 8]$. Finally, we can get three *min-wise* signatures of S_1 from $\pi(V_1)$ (i.e., 0, 5, 8) and three *min-wise* signatures of S_2 from $\pi(V_2)$ (i.e., 1, 6, 8).

| | bin 1 | | | | bin 2 | | | | bin 3 | | | |
|--------------|-------|---|---|---|-------|---|---|---|-------|---|----|----|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| $\pi(V_1)$: | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| $\pi(V_2)$: | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |

Figure 3. An example of one-permutation min-wise hashing.

3.4. The GPU-Aided AP-Tree⁺ Indexing

This section develops an indexing approach aided by GPU. First, we map an AP-tree⁺ into the GPU's memory to form a G-AP-tree⁺. Then, we employ a GPU-aided approach for set similarity join to accelerate approximate keyword matching of G-AP-tree⁺. Finally, a CPU-GPU data communication scheme is used for efficiently processing geo-textual streaming data with G-AP-tree⁺.

3.4.1. Data Structure for G-AP-tree⁺

Operations on matrices, vectors and arrays naturally suit for the GPU architecture [31]. Our design intends to use a variety of one-dimensional arrays to organize different components of AP-tree⁺ including *spatial node*, *keyword node*, *query node*, and ordered keyword trie. Furthermore, to effectively support approximate keyword matching of G-AP-tree⁺, we utilize a compact *characteristic matrix* in [16] to represent q-grams of all keywords in AP-tree⁺.

The structure of G-AP-tree⁺ is illustrated in Figure 4. As we can see in Figure 4, a *spatial node array*, a *keyword node array*, and a *query node array* are respectively used to store all *spatial nodes*, *keyword nodes* and *query nodes* in AP-tree⁺. The root node in AP-tree⁺ may be a *spatial node* or a *keyword node*, thus we define a root node in G-AP-tree⁺ as an array holds two index entries (i.e., index 1 and index 2) where index 1 points to the *spatial node array* and index 2 links to the *keyword node array*. The value of index 2 is set to -1 and the value of index 1 is an index of *spatial node array* if the root node is *spatial node*, while the value of index 1 is set to -1 and the value of index 2 is an index of *keyword node array* if the root node is *keyword node*.

For each *spatial node*, m cells are in turn input into the *spatial node array*. Each cell consists of one region reflects its spatial area and one index that points to one corresponding child. Since one child node in AP-tree may be one *spatial node*, *keyword node* or *query node*, the index in the *spatial node array* may represent one index in the spatial node array, keyword node array or query node array. Similarly, for each *keyword node*, l cuts are in sequence filled into the *keyword node array*. Since *keyword nodes* are generated based on an ordered keyword trie, we also map an ordered keyword trie into GPU memory. Concretely, we use m one-dimensional arrays to store m levels of one ordered keyword trie. Thus, every cut in the *keyword node array* holds a pair of <index, triple>. The index has the same function as the index in *spatial node array*. The triple is denoted as (L, W_s, W_e) , where L presents the level, W_s is the lower boundary keyword and W_e is the upper boundary keyword in the L -th level of ordered keyword trie. Furthermore, for each query node, n queries at most can be filled into a *query node array*. Each query holds two parts: a *region* that represents its spatial area and a *word list* contains a set of keywords. Noted that, all keywords in both *query node array* and *keyword node array* are represented by *min-wise* signatures as well.

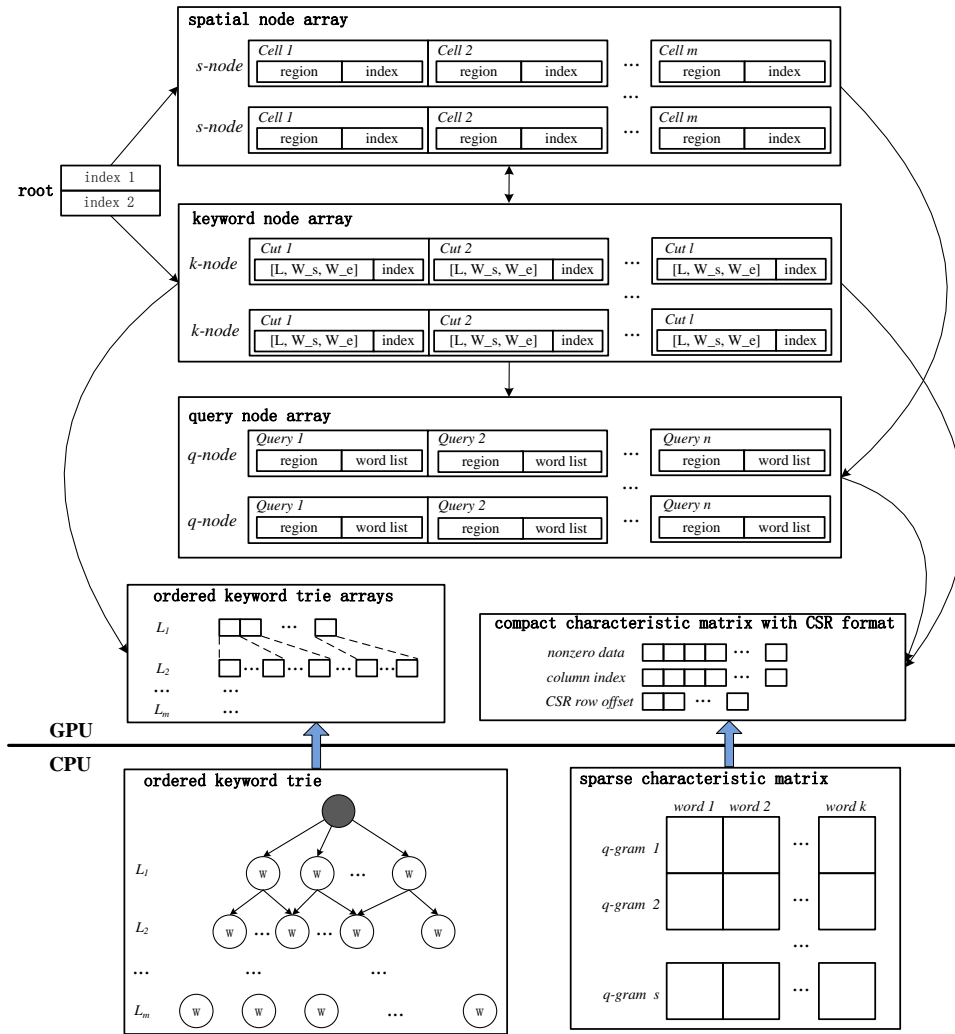


Figure 4. Illustrating the structure of G-AP-tree⁺.

To support approximate keyword matching, we employ a compact matrix structure *characteristic matrix* in [16]. As we can see in Figure 4, a characteristic matrix assigns the value as 1 when a q-gram represented by a row belongs to a keyword represented by a column, and 0 when it does not. Since the characteristic matrix is highly sparse, we employ Compressed Sparse Row (CSR) format mentioned in [16] to compress our characteristic matrix to fit the GPU memory. Noted that we have made order for all keywords and q-grams in advance. Thus, we only hold index of keywords and q-grams in the GPU memory. Meanwhile, we dynamically maintain the characteristic matrix based on the keywords in the query workload in host side, and then transfer the compact characteristic matrix into the GPU memory.

3.4.2. Parallelising Approximate Keyword Matching

Once the G-AP-tree⁺ is constructed in the GPU memory, we can constantly input data segments (see Definition 4) from a geo-textual data stream into GPU, and then use a G-AP-tree⁺ to filter data tuples in a segment in parallel.

Definition 4 (Segment). A segment, denoted as $g = \{s_1, \dots, s_m\}$, is a set of m spatial-textual tuples in timestamp order.

During the parallel computing procedure, however, the efficiency of similarity computing of keywords between queries and data tuples based on *min-wise* hashing is a problem. We proposed a parallel scheme for solving this issue. The parallel scheme is shown in Figure 5. For simplification, we assume that each data tuple s_i ($1 \leq i \leq m$) has two keywords. Thus, we first construct m *signature matrices* in parallel based on the first keywords of s_i and the keywords from the *characteristic matrix* of G-AP-tree⁺. Each *signature matrix* stores *min-wise* signatures of q-grams of keywords from both the streaming data tuple and queries in G-AP-tree⁺. All *signature matrix* are constructed in parallel as well. The construction procedure of one *signature matrix* is shown in Algorithm 1.

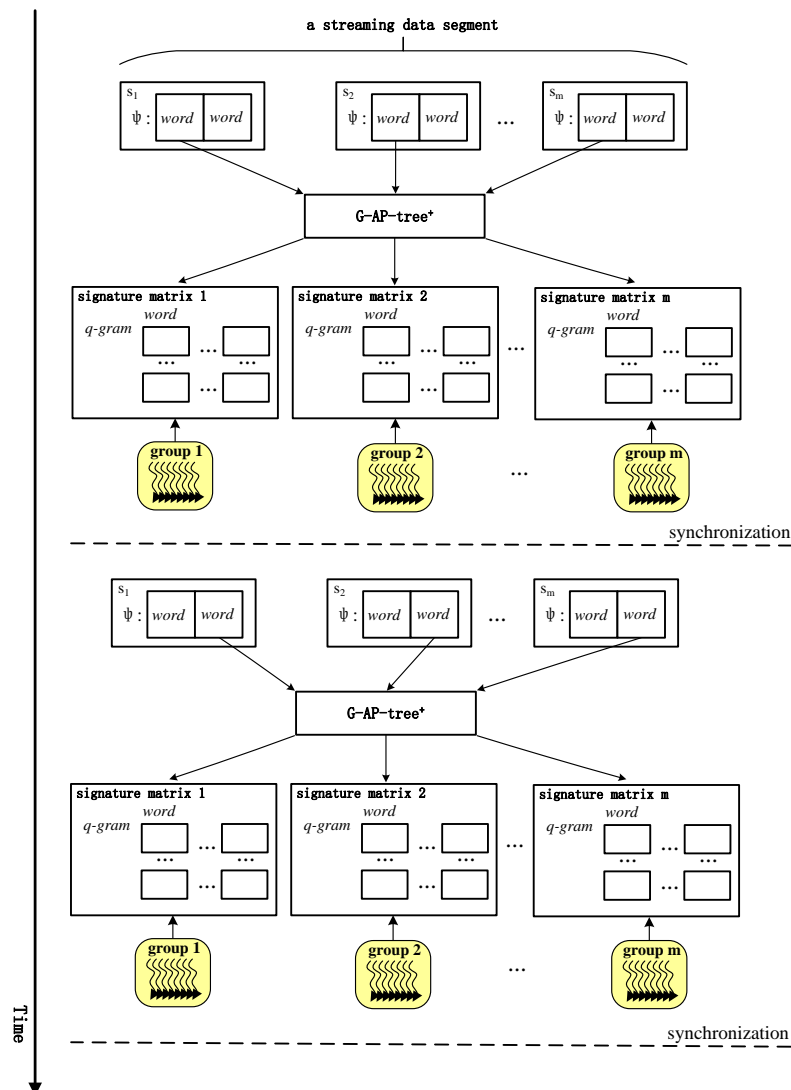


Figure 5. Illustrating the parallel scheme of approximate keyword matching.

Algorithm 1: Constructing a signature matrix on GPU

-
- ```

1 Construction_Procedure (w, T) /* Input : w is one keyword of data tuple, T is a
 G-AP-tree+ */
/* Output: M is a signature matrix */
/* Construct a q-gram-keyword matrix */
2 Retrieve n min-wise signatures of keywords from T for matching w .
3 Transfer n min-wise signatures into n q-gram-keyword vectors (v_1, v_2, \dots, v_n) with the
 characteristic matrix.
4 Compute the min-wise signature of w and transfer it to a q-gram-keyword vector l with the
 characteristic matrix.
5 Assemble one signature matrix $M = \{l, v_1, v_2, \dots, v_n\}$.

```
- 

Furthermore, we can simultaneously execute the approximate matching of signatures between the first keyword of the streaming data tuple and the keywords of queries in G-AP-tree<sup>+</sup> by computing the similarity among the *min-wise* signatures in one *signature matrix*. After then, we execute the same procedure for the second keywords of all streaming data tuples.

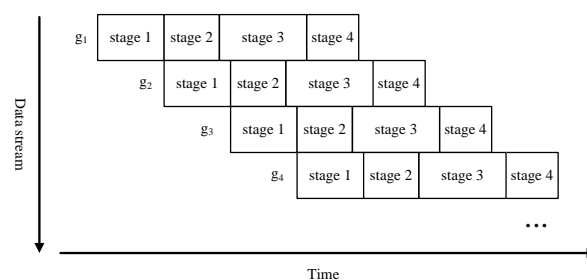
### 3.4.3. A Communication Scheme for Processing Geo-Textual Streaming Data with G-AP-Tree<sup>+</sup>

In our setting, a G-AP-tree<sup>+</sup> is utilized to continuously filter data segments from a geo-textual data stream  $S$ . However, some issues complicate the efficient use of GPU for the streaming data filtering. First, CPU and GPU have separate memories, we have to explicitly partition the streaming data into segments and copy them into the GPU memory. Nevertheless, the efficient partitioning is not always straight-forward. Then, the PCI-E link that connects the two memories has limited bandwidth can often be a bottleneck, starving GPU cores from their data. Finally, the high bandwidth of GPU memory can only be exploited when GPU threads executing at the same time access memory in a coalesced fashion, where the threads simultaneously access adjacent memory locations. For efficient streaming data filtering, we applied “*BigKernel*” [17] that is a data communication scheme between CPU and GPU to address the above issues. *BigKernel* can use a four-stage pipeline with an automated prefetching method to (i) optimize CPU-GPU communication and (ii) optimize GPU memory accesses.

In our setting, to filter a segment  $g$ , which is represented by an array, the four-stage pipeline is as following:

1. Prefetch address generation: transforming the read accesses to the  $g$  array to instead store the addresses in a CPU-side address buffer
2. Data assembly: using a CPU thread to fetch the corresponding data element from the  $g$  array for each address in the address buffer and placing it in one prefetch buffer, which also must be a pinned buffer.
3. Data transfer: the data transfer stage is executed by the GPU streaming engine, transferring data from CPU-side prefetch buffer to GPU-side data buffer.
4. Kernel computation: filtering data tuples in data buffer using G-AP-tree<sup>+</sup>.

For a data stream  $S = \{g_1, g_2, g_3, g_4, \dots\}$ , the four-stage pipeline is illustrated by Figure 6.



**Figure 6.** Illustrating the communication scheme of the four-stage pipeline.

## 4. Results and Discussion

We first provided an experimental setup, then evaluated and discussed the performances of AP-Tree<sup>+</sup>, and the GPU-aided AP-Tree<sup>+</sup> against geo-textual streaming data using a NVIDIA GPU.

### 4.1. Experimental Setup

The data sets used in this paper come from TWEETS [13] and AIS. TWEETS is a real-life dataset collected from Twitter. The dataset contains 12 million tweets with geo-textual information from May 2012 to August 2012. AIS holds the geo-locations from Chorochronos Archive (<http://www.chorochronos.org>) and keywords from Newsgroups (<http://people.csail.mit.edu/ljrennie120Newsgroups>). The statistics of two datasets are summarized in Table 1.

**Table 1.** Datasets.

| Set Name | # of Objects | Vocabulary | Average # of Keywords in Objects |
|----------|--------------|------------|----------------------------------|
| TWEETS   | 12.7 M       | 1.7 M      | 9                                |
| AIS      | 5.7 M        | 81 K       | 50                               |

For query workload, like [9], we randomly select 5M geo-textual objects from the dataset. For each selected object,  $k$  terms are randomly picked as query keywords and  $k$  is a random number between 1 and 5. The query region is set to a rectangle, and the region size is uniformly chosen between 0.01% and 1% of the universe data space.

All experiments were executed on one computer equipped with a GPU (GTX TITAN X), and the configurations are presented in Table 2.

**Table 2.** Configurations of the computer.

| Specifications of CPU Platforms | Computer                            |
|---------------------------------|-------------------------------------|
| OS                              | Ubuntu14.04                         |
| CPU                             | i7-5820k (3.3 GHz, 6 cores)         |
| Memory                          | 32 GB DDR4                          |
| Specifications of GPU Platforms | GTX TITAN X                         |
| Architecture                    | Maxwell                             |
| Memory                          | 12 GB DDR5                          |
| Bandwidth                       | Bi-directional bandwidth of 16 GB/s |
| CUDA                            | SDK 7.0                             |

### 4.2. Evaluating and Discussing AP-Tree<sup>+</sup>

In this section, we evaluate space cost, index maintenance cost and query performance of AP-tree<sup>+</sup> over a geo-textual data stream.

For comparison, we used a state-of-the-art indexing structure MHR-tree [14] due to the reason that MHR-tree can support spatial approximate keyword queries. We directly employ the MHR-tree to process continuous spatial approximate keyword queries in our setting. For AP-tree<sup>+</sup>, the partition threshold  $\theta_q$  and the fanout factor  $f$  are set to 200 and 40 respectively, which are better trade-off between index size and matching performance according to the experimental results in [9]. Meanwhile, for both AP-tree<sup>+</sup> and MHR-tree, the q-gram length is 2 and the edit distance threshold is  $\tau = 2$ .



#### 4.2.1. Space Costs

In this experiment, we investigate storage cost of AP-tree<sup>+</sup>. AP-tree and MHR-tree are used for comparison. For MHR-tree, the number of *min-wise* hashing functions  $\ell = 50$  according to the experiments in [14], while the AP-tree<sup>+</sup> used one-permutation *min-wise* hashing method.

As shown in Figures 7a and 8a, both AP-tree<sup>+</sup> and MHR-tree take more storage overheads than AP-tree, due to the additional min-wise signatures. Moreover, as the number of queries  $N$  increases from 1 M to 2 M, 3 M, 4 M, and 5 M, the storage cost of AP-tree<sup>+</sup> is respectively about 88.5% for TWEETS and 89.9% for AIS of MHR-tree, on average. The major reason of space saving lies in that we only need to embed signatures into the  $q$ -nodes and  $k$ -nodes of AP-tree<sup>+</sup> while MHR-tree has to append signatures for all nodes. The Figures 7b and 8b can illustrate the fact. As we can see, the size of signatures of AP-tree<sup>+</sup> is about 78.9% and 79.8% of one of MHR-tree, respectively for TWEET and AIS datasets.

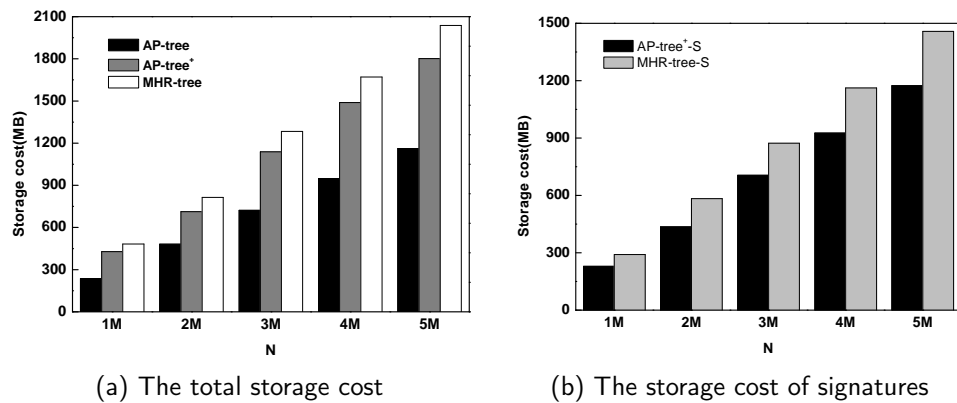


Figure 7. The space cost of AP-tree<sup>+</sup> based on TWEET dataset.

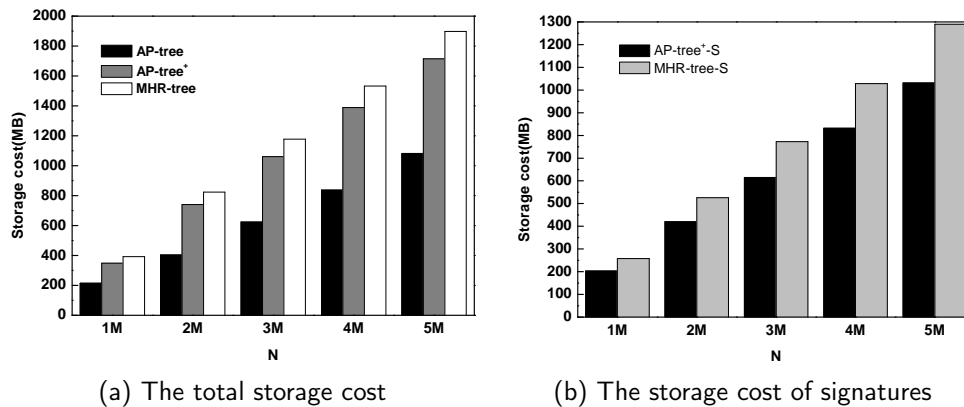


Figure 8. The space cost of AP-tree<sup>+</sup> based on AIS dataset.

#### 4.2.2. Maintenance Overheads

In the experiment, we compared the maintenance costs between AP-tree<sup>+</sup> and MHR-tree. We built an AP-tree<sup>+</sup> and a MHR-tree based on datasets from TWEETS and AIS, respectively. Then we maintain the two indexing structures by updating datasets. One update is treated as a combination of a separate deletion and insertion. We first measure the CPU runtime incurred by updating various numbers of queries in indexing structures.

Figure 9 shows that AP-tree<sup>+</sup> outperforms MHR-tree in terms of maintenance cost. As  $N$  changes from 1 M to 5 M, the ratio of update time of AP-tree<sup>+</sup> to the one of MHR-tree is 53.15% and 63.52% on average, respectively for TWEETS and AIS. The great performance gains lie in that MHR-tree used  $\ell = 50$  *min-wise* hashing functions to generate signatures while AP-tree<sup>+</sup> only employed one hashing function to do so during the update procedure of indexing structure.

We also compared the maintenance cost of AP-tree<sup>+</sup> with MHR-tree using various number of *min-wise* hashing functions, by updating a set of 500K queries. Figure 10a shows that the ratio of update time of AP-tree<sup>+</sup> to the one of MHR-tree is 59.0%, 39.4%, 27.6%, 21.3%, 16.9%, and 14.6% for TWEETS, as the number of functions increases from 50, 60, 70, 80, 90, to 100. The experimental result indicates that AP-tree<sup>+</sup> has an obvious superiority over MHR-tree in the case that the number of hash functions is large. The reason is that the cost of updating one signature for MHR-tree is multiplied by the number of *min-wise* hashing functions, while the one for AP-tree<sup>+</sup> retains a constant due to the employment of one-permutation hashing function. Figure 10b reflects the similar results.

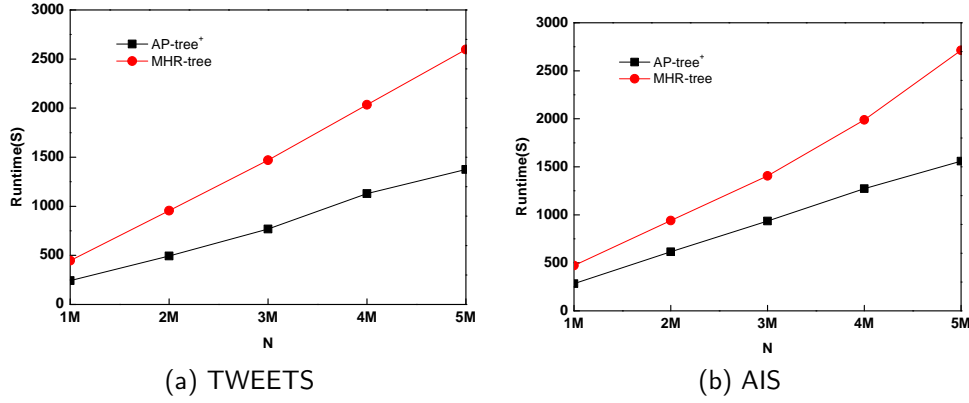


Figure 9. The maintenance cost of AP-tree<sup>+</sup> with various numbers of queries.

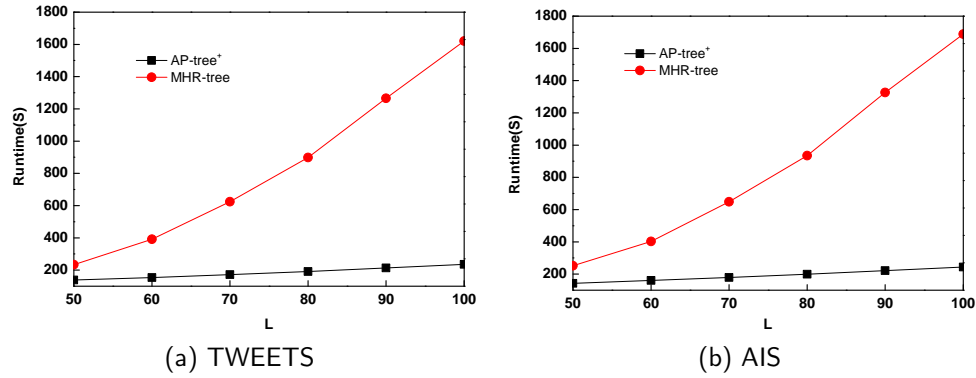


Figure 10. The maintenance cost of AP-tree<sup>+</sup> comparing with MHR-tree with various numbers of *min-wise* hashing functions.

#### 4.2.3. Query Performance

Like in [14], we also first employ the evaluation metric *recall* to measure the query accuracy of AP-tree<sup>+</sup>. Then, we evaluate the CPU *runtime* of AP-tree<sup>+</sup> filtering streaming data. The MHR-tree is employed as one baseline. We use a TWEETS query set  $Q$  with  $|Q| = 500$  K for these experiments. Since we run multiple continuous queries over a data stream with  $N$  data tuples in our case, the *recall* is defined as Formula (7).

$$recall = \frac{\sum_{i=1}^N a_i^x}{\sum_{i=1}^N a_i} \quad (7)$$

where,  $a_i$  is the number of continuous queries that the  $i$ th streaming data tuple correctly satisfied and  $a_i^x$  is the number of continuous queries returned by AP-tree<sup>+</sup> or MHR-tree for the  $i$ th streaming data tuple.

We first compared the recall of AP-tree<sup>+</sup> with the one of MHR-tree with 50 *min-wise* hashing functions over a data stream (see Figure 11a). Then, given a streaming data segment whose size = 1 M, we also measured the recalls of both AP-tree<sup>+</sup> and MHR-tree with various numbers of *min-wise* hashing functions (see Figure 11b, the number of *min-wise* hashing functions  $L$  ranges from 50, 60, 70,

80, 90, to 100). Figure 11a,b show that AP-tree<sup>+</sup> has one comparable recall with the one of MHR-tree. The reason is that the one-permutation-based *min-wise* hashing can achieve similar or even better accuracies compared with the multi-permutations-based *min-wise* hashing algorithm.

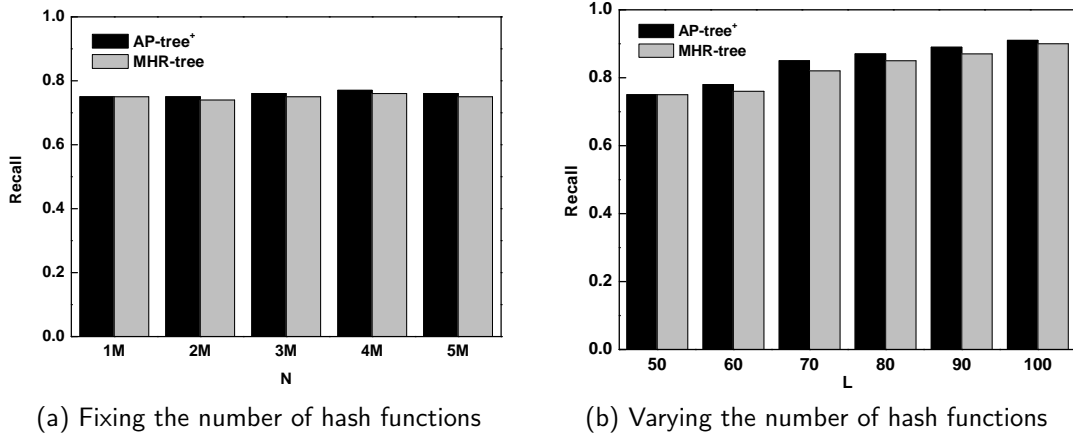


Figure 11. The recall of AP-tree<sup>+</sup> comparing with MHR-tree.

Meanwhile, we measured CPU runtime of filtering a stream with N data tuples using AP-tree<sup>+</sup> and MHR-tree. S means second in Figure 12. Figure 12 reflects that AP-tree<sup>+</sup> can achieve about 5.64× performance gain comparing to MHR-tree for various sizes of streaming data. The reason for such experimental results lies in that AP-tree<sup>+</sup> has a much better spatial-textual filtering capability due to inheriting the adaptability to spatial-textual query workload from AP-tree, while the construction of MHR-tree is based on the spatial-first scheme.

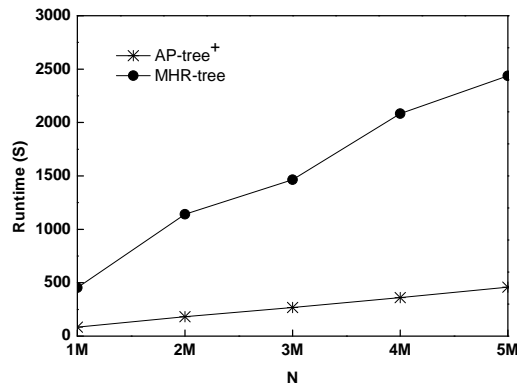


Figure 12. The CPU runtime of AP-tree<sup>+</sup> comparing with MHR-tree.

#### 4.3. Evaluating and Discussing the GPU-Aided AP-Tree<sup>+</sup>

For GPU-aided AP-tree<sup>+</sup>, we made three observations: (1) the space cost of G-AP-tree<sup>+</sup>, (2) the query performance of G-AP-tree<sup>+</sup> with the algorithm of parallel keyword approximate matching when handling streaming data, and (3) the impact of the communication scheme on query performance of G-AP-tree<sup>+</sup>.

##### 4.3.1. Space Costs

In this experiment, we investigate the storage cost of G-AP-tree<sup>+</sup> based on TWEETS dataset. According to the structure of G-AP-tree<sup>+</sup> in Figure 4, we can know that the space cost of one G-AP-tree<sup>+</sup> consists of three parts:

1. Node arrays—a root node array, a keyword node array, a spatial node array, and a query node array.
2. Ordered keyword trie arrays— $m$  one-dimensional arrays to store  $m$  levels of one ordered keyword trie.
3. One characteristic matrix—a compact characteristic matrix with CSR format.

Thus, we observed the GPU memory overheads of these three parts, respectively, for various sizes of query workloads. Figure 13 shows that part 1, part 2, and part 3 averagely account for 85.5%, 4.0% and 10.5% of the total storage cost, respectively. That means we need about 14.5% auxiliary space overhead to map G-AP-tree<sup>+</sup> in the GPU memory.

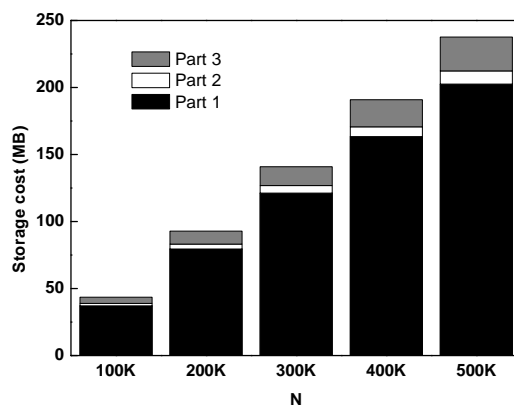


Figure 13. Evaluating the space cost of G-AP-tree<sup>+</sup>.

#### 4.3.2. Query Performance

We made the following observation: the query performance of G-AP-tree<sup>+</sup> with the algorithm of parallel keyword approximate matching when handling streaming data. We map an AP-tree<sup>+</sup> with a TWEETS query set  $Q$  whose size  $|Q| = 500$  K into the GPU memory to form a G-AP-tree<sup>+</sup>, and then investigate the *runtime* of filtering a stream with  $N$  data tuples. According to the experimental results in Figure 14, as the number of streaming data  $N$  increases, G-AP-tree<sup>+</sup> gains an average speedup of  $5.76\times$  than AP-tree<sup>+</sup>. The results indicate that the GPU-aided indexing method has been useful for significantly accelerating the procedure of filtering geo-textual streaming data.

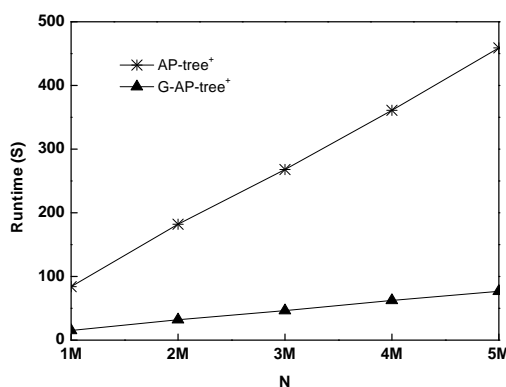


Figure 14. Evaluating the query performance of G-AP-tree<sup>+</sup>.

#### 4.3.3. The Effect of Communication Scheme

In this experiment, we measure time overhead of G-AP-tree<sup>+</sup> with the communication scheme (denoted as G-AP-tree<sup>+</sup>+comm). We again map an AP-tree<sup>+</sup> with a TWEETS query set  $Q$  whose size  $|Q| = 500$  K into the GPU memory to form a G-AP-tree<sup>+</sup>. Then, we compared the *runtime* of filtering a stream with  $N$  data tuples of G-AP-tree<sup>+</sup> and G-AP-tree<sup>+</sup>+comm. According to the

experimental results in Figure 15, as the number of streaming data  $N$  increases, G-AP-tree<sup>+</sup>+comm can improve the query performance by 39.4% compared to G-AP-tree<sup>+</sup>. The reasons that the communication optimization can accelerate the procedure of filtering streaming data are because of overlap between computation and data communications, the volume reduction of CPU-GPU data communications and coalesced accesses to GPU memory by placing the input data of consecutive threads in interleaved data segments.

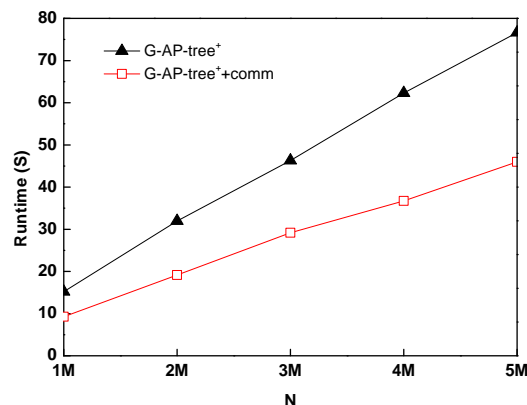


Figure 15. Evaluating the query performance of G-AP-tree<sup>+</sup> with the communication scheme.

## 5. Conclusions

It is a basic requirement for many current applications to support continuous spatial-keyword queries. However, existing spatial-keyword query indexing approaches generally do not apply for approximate keyword matching. To address this problem, this paper first proposes an indexing approach for efficient supporting of continuous spatial approximate keyword queries by integrating *min-wise* signatures to AP-tree, namely AP-tree<sup>+</sup>. AP-tree<sup>+</sup> can employ one-permutation *min-wise* hashing method to efficiently support approximate keyword matching. Towards providing a more efficient indexing approach, this paper has explored the feasibility of paralleling AP-tree<sup>+</sup> with GPU. The experimental results show that (1) AP-tree<sup>+</sup> can reduce the space cost by about 11% compared with MHR-tree, (2) AP-tree<sup>+</sup> can hold a comparable recall and 5.64× query performance gain compared with MHR-tree while saving 41.66% maintenance cost on average, (3) the GPU-aided AP-tree<sup>+</sup> was 5.76× faster on average than AP-tree<sup>+</sup>, and (4) the GPU-CPU data communication scheme could further improve the query performance of GPU-aided AP-tree<sup>+</sup> by 39.4%. At present, our approach for continuous spatial approximate keyword queries only focuses on a geo-textual data stream. In the future, we plan to extend our approach to distributed geo-textual data streams based on Apache Storm or Spark.

**Author Contributions:** Conceptualization, Z.D. and L.W.; methodology, Z.D. and M.W.; software, J.C. and X.H.; validation, W.H.; formal analysis, Z.D.; investigation, J.C.; resources, L.W.; writing—original draft preparation, Z.D. and L.W.; writing—review and editing, L.W.; supervision, A.Y.Z.; project administration, L.W.

**Funding:** This work is supported in part by the National Natural Science Foundation of China (No. U1711266), the National Science and Technology Major Project of the Ministry of Science and Technology of China (2016ZX05014-003), the China Postdoctoral Science Foundation (2014M552112), the Fundamental Research Funds for the National University, China University of Geosciences(Wuhan).

**Conflicts of Interest:** The authors declare no conflict of interest.



## Abbreviations

The following abbreviations are used in this paper:

|                              |                                                         |
|------------------------------|---------------------------------------------------------|
| $t$                          | a geo-textual tuple                                     |
| $t.\phi$                     | a set of keywords for tuple $t$                         |
| $t.loc$                      | a geo-location for tuple $t$                            |
| $t.ts$                       | a timestamp to label the creation time of tuple $t$     |
| $S$                          | a geo-textual data stream                               |
| $q$                          | a continuous spatial approximate keyword query          |
| $q.\psi$                     | a set of keywords for query $q$                         |
| $q.r$                        | a range region for query $q$                            |
| $\text{sim}(t.\phi, q.\psi)$ | the similarity between $t.\phi$ and $q.\psi$            |
| $\tau$                       | a similarity threshold $\in [0, 1]$                     |
| $Q$                          | a set of continuous spatial approximate keyword queries |

## References

1. Zhong, X.; Kealy, A.; Sharon, G.; Duckham, M. Spatial interpolation of streaming geosensor network data in the RISER system. *LNCS* **2015**, *9080*, 161–177.
2. Galić, Z.; Baranović, M.; Križanović, K.; Mešković, E. Geospatial data streams: Formal framework and implementation. *Data Knowl. Eng.* **2014**, *91*, 1–16. [\[CrossRef\]](#)
3. Eom, S.; Shin, S.; Lee, K.-H. Spatiotemporal query processing for semantic data stream. In Proceedings of the 2015 IEEE 9th International Conference on Semantic Computing (IEEE ICSC 2015), Anaheim, CA, USA, 7–9 February 2015; pp. 290–297.
4. Galić, Z. *Spatio-Temporal Data Streams and Big Data Paradigm*; Springer: New York, NY, USA, 2016.
5. Galić, Z.; Mešković, E.; Osmanović, D. Distributed processing of big mobility data as spatio-temporal data streams. *Geoinformatica* **2017**, *21*, 263–291. [\[CrossRef\]](#)
6. Brandt, T.; Grawunder, M. Geostreams: A survey. *ACM Comput. Surv.* **2018**, *51*, 1–37. [\[CrossRef\]](#)
7. Chen, L.; Cong, G.; Cao, X. An efficient query indexing mechanism for filtering geo-textual data. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 22–27 June 2013; pp. 749–760.
8. Chen, X.; Zhang, D.; Wang, L.; Jia, N.; Kang, Z.; Zhang, Y.; Hu, S. Design automation for interwell connectivity estimation in petroleum cyber-physical systems. *IEEE Trans. CAD Integr. Circuits Syst.* **2017**, *36*, 255–264. [\[CrossRef\]](#)
9. Wang, X.; Zhang, Y.; Zhang, W.; Lint, X.; Wang, W. Ap-tree: Efficiently support continuous spatial-keyword queries over stream. In Proceedings of the 2015 IEEE 31st International Conference on Data Engineering (ICDE), Seoul, Korea, 13–17 April 2015; pp. 1107–1118.
10. Babu, S.; Widom, J. Continuous queries over data streams. *SIGMOD Rec.* **2001**, *30*, 109–120. [\[CrossRef\]](#)
11. Golab, L.; Özsu, M.T. Issues in data stream management. *SIGMOD Rec.* **2003**, *32*, 5–14. [\[CrossRef\]](#)
12. Park, J.; Hong, B.; Ban, C. A continuous query index for processing queries on RFID data stream. In Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Daegu, Korea, 21–24 August 2007; pp. 138–145.
13. Li, G.; Wang, Y.; Wang, T.; Feng, J. Location-aware publish/subscribe. In Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, IL, USA, 11–14 August 2013; pp. 802–810.
14. Li, F.; Yao, B.; Tang, M.; Hadjieleftheriou, M. Spatial approximate string search. *IEEE Trans. Knowle. Data Eng.* **2013**, *25*, 1394–1409. [\[CrossRef\]](#)
15. Li, P.; Owen, A.; Zhang, C.H. One permutation hashing. In Proceedings of the Advances in Neural Information Processing Systems 25, Lake Tahoe, NV, USA, 3–6 December 2012; pp. 3113–3121.
16. Cruz, M.S.; Kozawa, Y.; Amagasa, T.; Kitagawa, H. Gpu acceleration of set similarity joins. *LNCS* **2015**, *9261*, 384–398.
17. Mokhtari, R.; Stumm, M. Bigkernel—High performance cpu-gpu communication pipelining for big data-style applications. In Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS), Phoenix, AZ, USA, 19–23 May 2014; pp. 819–828.

18. Deng, Z.; Wang, L.; Chu, J.; Huang, X.; Han, W.; Zomaya, A.Y. An indexing approach for efficient supporting of continuous spatial approximate keyword queries. In Proceedings of the 20th IEEE International Conference on High Performance Computing and Communications (HPCC), Exeter, UK, 28–30 June 2018; pp. 132–139.
19. Göbel, R.; Henrich, A.; Niemann, R.; Blank, D. A hybrid index structure for geo-textual searches. In Proceedings of the 18th ACM conference on Information and knowledge management (CIKM), Hong Kong, China, 2–6 November 2009; pp. 1625–1628.
20. Zhang, D.; Chee, Y.M.; Mondal, A.; Tung, A.K.H.; Kitsuregawa, M. Keyword search in spatial databases: Towards searching by document. In Proceedings of the 2009 IEEE 25th International Conference on Data Engineering (ICDE), Shanghai, China, 29 March–2 April 2009; pp. 688–699.
21. Zhang, C.; Zhang, Y.; Zhang, W.; Lin, X. Inverted linear quadtree: Efficient top k spatial keyword search. In Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE), Brisbane, QLD, Australia, 8–12 April 2013; pp. 901–912.
22. Li, J.; Wang, H.; Li, J.; Gao, H. Skyline for geo-textual data. *Geoinformatica*, **2016**, *20*, 453–469. [\[CrossRef\]](#)
23. Deng, D.; Li, G.; Feng, J. A pivotal prefix based filtering algorithm for string similarity search. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, Snowbird, UT, USA, 22–27 June 2014; pp. 673–684.
24. Li, C.; Wang, B.; Yang, X. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB), Vienna, Austria, 23–27 September 2007; pp. 303–317.
25. Li, C.; Lu, J.; Lu, Y. Efficient merging and filtering algorithms for approximate string searches. In Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE), Cancun, Mexico, 7–12 April 2008; pp. 257–266.
26. Zhang, Z.; Hadjieleftheriou, M.; Ooi, B.C.; Srivastava, D. Bed-tree: An all-purpose index structure for string similarity search based on edit distance. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, Indianapolis, IN, USA, 6–10 June 2010; pp. 915–926.
27. Broder, A.Z.; Charikar, M.; Frieze, A.M.; Mitzenmacher, M. Min-wise independent permutations. *J. Comput. Syst. Sci.* **2000**, *60*, 630–659. [\[CrossRef\]](#)
28. Li, P.; Shrivastava, A.; König, A.C. Gpu-based minwise hashing. In Proceedings of the 21st International Conference on World Wide Web (WWW), Lyon, France, 16–20 April 2012; pp. 565–566.
29. Shrivastava, A.; Li, P. Densifying one permutation hashing via rotation for fast near neighbor search. In Proceedings of the International Conference on Machine Learning, Beijing, China, 21–26 June 2014; pp. 557–565.
30. Deng, Z.; Wu, X.; Wang, L.; Chen, X.; Ranjan, R.; Zomaya, A.; Chen, D. Parallel processing of dynamic continuous queries over streaming data flows. *IEEE Trans. Parallel Distrib. Syst.*, **2015**, *26*, 834–846. [\[CrossRef\]](#)
31. Zhou, K.; Hou, Q.; Wang, R.; Guo, B. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.* **2008**, *27*, 126–136. [\[CrossRef\]](#)

