A Generic Inverted Index Framework for Similarity Search on the GPU

Jingbo Zhou[†], Qi Guo[†], H. V. Jagadish[#], Wenhao Luan[†],
Anthony K. H. Tung[†], Yueji Yang[†], Yuxin Zheng[†]
[†]School of Computing, National University of Singapore
[†]{jzhou, qiguo, luan1, atung, yueji, yuxin}@comp.nus.edu.sg
[#] Univ. of Michigan, Ann Arbor
[#]jag@umich.edu

ABSTRACT

To perform similarity search, existing works mainly choose to create customized indexes for different data types. Due to the diversity of customized indexes, it is hard to devise a general parallelization strategy to speed up the search. In this paper, we propose a generic inverted index framework on the GPU (called GENIE), which can support parallel similarity search of multiple queries on various data types. We also give an in-depth investigation to demonstrate how to properly fit both the *Locality Sensitive Hashing* scheme and the *Shotgun and Assembly* scheme to our framework for similarity search on the GPU. Extensive experiments on different real-life datasets demonstrate the efficiency and effectiveness of our framework. The implemented system has been released as open source[1].

1. INTRODUCTION

The Big Data revolution has resulted in the generation of a large variety of complex data in the form of documents, sets, sequences, trees, graphs and high dimensional vectors [49]. To perform similarity search on these data efficiently, a widely adopted approach is to create indexes. More often than not, the proposed index also varies for different complex data types [46, 37, 58, 63, 64, 10]. Parallelization is required for high performance on modern hardware architectures. Since each type of customized index has its own properties and structures, different parallelization strategies must be adopted for each type of index [31, 61, 32, 12, 57]. It is desirable to design a generic index framework for parallel similarity search on modern hardware architectures.

Our insight for generic index framework is that many data types can be transformed into a form, which can be searched by an inverted-index-like structure. Such transformation can be done by the *Locality Sensitive Hashing* (LSH) scheme under several similarity measures [11, 14] or by the *Shotgun and Assembly* (SA) [5, 52] scheme for complex structured data like documents, sequences, trees and graphs. To support our claim, let us first illustrate how the inverted index can be applicable for similarity search on a variety of complex data types after some transformations.

- **Transformed by LSH.** The most common data type, **high dimensional data**, can be transformed by an LSH scheme [14] derived from the $\mathfrak{p}$-stable distribution in $l_{\mathfrak{p}}$ space. In such a scheme, multiple hash functions are used to hash data points into different buckets and points that are frequently hashed to the same bucket are deemed to be similar. Hence we can build an inverted index where each posting list corresponds to a list of points hashed to a particular bucket. Given a query point, Approximate Nearest Neighbour (ANN) search can be performed by first hashing the query point and then scanning the corresponding posting list to retrieve data points that appear in many of these buckets. Meanwhile, **Sets, feature sketches and geometries** typically have kernelized similarity functions. These includes Jaccard kernel for sets [36], Radial Basis Function (RBF) kernel for feature sketches, and Geodesic kernel for hyperplanes [11]. Similarity search on such data can also be transformed by Locality Sensitive Hashing functions [11] and searched through an inverted index.

- **Transformed by SA**. The complex structure data, including **documents, sequences, trees and graphs**, can be transformed with the Shotgun and Assembly [5, 52] scheme. Specifically, the data will be broken down into smaller sub-units ("shotgun"), such as words for documents [56, 43], q-grams for sequences [37, 61], binary branches for trees [64] and stars for graph [63, 60]. After the decomposition, we can build an inverted index with a posting list for each unique sub-unit and data objects containing a particular sub-unit are stored in the posting list. During query time, query objects will also be broken down into a set of smaller sub-units and the corresponding posting lists will be accessed to find data objects that share a lot of common sub-units with the query object. This approach has been widely used for similarity search of complex structured data [61, 64, 63, 58]. Besides, the inverted index for **relational data** can also be built

[1]https://github.com/SeSaMe-NUS/GPUGENIE-Release

using the similar SA scheme. We first decompose each tuple into a set of attribute-value pairs. A posting list is then allocated to each unique attribute-value pair to store tuples that contain this particular pair. Given a query, the similarity search can be done by scanning the posting lists of corresponding attribute-value pairs. For attributes with continuous value, we assume that they can be discretized to an acceptable granularity level.

Though it is intuitive to build inverted index for different data types, to the best of our knowledge, there is no existing generic inverted index framework for similarity search. As discussed above, with similar index methods, the operation on the index is not the same for different data types. The problem is to propose a unified model processing the inverted index for different data types. Specially, the model should be able to support both the LSH and SA scheme.

Aside from the usefulness of an inverted index in similarity search, we also observe that very often, it is useful to answer multiple similarity queries in parallel. For example, image matching is often done by extracting hundreds of high dimensional[2] SIFT (scale-invariant feature transform) features and matching them against SIFT features in the database [9]. As another example, in sequence search, users might be interested in similar subsequences that are within the query sequence. In such a case, the query sequence will be broken down into multiple subsequences on which multiple similarity search will be invoked. To sum up, it is desirable to design a general-purpose index that can be scaled up to a large number of similarity search queries being submitted in parallel.

To effectively parallelize the operations of our proposed framework, we choose to implement it on the Graphics Processing Units (GPUs). GPUs have experienced a tremendous growth in terms of computational power and memory capacity in recent years. One of the most advanced GPU in the consumer market, the Nvidia GTX Titan X, has 12 GB of DDR5 memory at a price of 1000 US dollars while an advanced server class GPU, the Nvidia K80, has 24GB of DDR5 memory at a price of 5000 US dollars. The recently released Nvidia Pascal GPU is with 16GB of HBM2 memory and 720 GB/s peak bandwidth, three times the current bandwidth [3]. Furthermore, most PCs allow two to four GPUs to be installed, bringing the total amount of GPU memory in a PC to be compatible with an in-memory database node [67]. More importantly, the GPU's parallel architecture – SIMD on massive number of small cores – suits our processes on the inverted index which are mostly simple matching, scanning and counting.

In view of the above considerations, we propose a Generic Inverted Index framework, called GENIE, which is implemented on the GPU to support similarity search in parallel (we also name our system as GENIE). To cater for a variety of data types, we introduce an abstract model, named match-count model, which can be instantiated to specific models for different data types. We will showcase that the match-count model can support both the LSH scheme and the SA scheme for similarity search using GENIE.

The design of GENIE is on the basis of a careful analysis of architecture characters of the GPU. GENIE is designed to support efficient and parallel computation of the match-count model for multiple queries on the GPU. The query processing of GENIE essentially performs scan on posting lists that are relevant to the queries and maintains a frequency count for the number of times that a data object is seen in these posting lists. Especially, we propose a novel data structure on the GPU, called Count Priority Queue (c-PQ for short), which can not only reduce the time cost for similarity search, but also reduce the memory requirement for multiple queries. The deployment of c-PQ can substantially increase the number of queries within a query processing batch on the GPU.

We also formally explore the theoretical bound for ANN search under the LSH scheme based on our match-count model. We first introduce the concept of Tolerance-Approximate Nearest Neighbour (τ-ANN) search for LSH in the same spirit of the popular c-ANN search [26]. Then we prove that GENIE can support the τ-ANN search for any similarity measure who has a generic LSH scheme. For complex data types without known LSH transformation, there is a choice of adopting the SA scheme. We will also showcase this by performing similarity search on sequence data, short document data and relational data using GENIE.

We summarize our contributions as follows:

- We propose a generic inverted index framework (GENIE) on the GPU, which can support similarity search of multiple queries under for different data types on the GPU under LSH scheme or SA scheme.
- We present the system design of the inverted index on the GPU. We also devise a novel data structure, named c-PQ, which significantly increases the throughput for multi-query processing on the GPU.
- We introduce a new concept for ANN search, the τ-ANN search, and demonstrate that GENIE can effectively support the τ-ANN search under the LSH scheme.
- We showcase the similarity search on complex data structures in original space by GENIE under the SA scheme.
- We conduct comprehensive experiments on different types of real-life datasets to demonstrate the effectiveness and efficiency of GENIE.

The rest of the paper is organized as follows. First, we will discuss related work in Section 2. Next, we will present a framework overview and preliminary definitions of this paper in Section 3. Then we will expound the system design of our index in Section 4, followed by a discussion about the similarity search on GENIE in Section 5 and Section 6. Finally, we will conduct experiment evaluation in Section 7 and will conclude the paper in Section 8.

2. RELATED WORK

2.1 Similarity search on complex Data

2.1.1 High dimensional data

Due to the "curse of dimensionality", spatial index methods, like R tree [20], R+ tree [51] and R* tree [8], provide

[2]128 dimensions to be exact

[3]https://devblogs.nvidia.com/parallelforall/inside-pascal/

little improvement over a linear scan algorithm when dimensionality is high. It is often unnecessary to find the exact nearest neighbour, leading to the development of ANN search in high dimensional space. The theoretical foundation of ANN search is based on the LSH family, which is a family of hash functions that map similar points into the same buckets with a high probability [26]. An efficient LSH family for ANN search in high dimensional space is derived from the $\mathfrak{p}$-stable distribution [14]. We refer interested readers to a survey of LSH [59].

2.1.2 Sets, feature sketches and geometries

The similarity between these objects is often known only implicitly, so the computable kernel function is adopted for similarity search. For example, the Jaccard kernel distance is used to estimate the similarity between two sets as $sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$. To scale up similarity search on these objects, the ANN search in such kernel spaces has also drawn considerable attention. The most notably solution for this problem is the LSH. For example, Charikar [11] investigates several LSH families for kernelized similarity search. Kulis and Grauman [33, 34] propose a data-dependent method to build LSH family in a kernel space. Wang et al. [59] give a good survey about the LSH scheme on different data types. GENIE can support the similarity search in an arbitrary kernel space if it has an LSH scheme.

2.1.3 Documents, sequences, trees and graphs

There is a wealth of literature concerning the similarity search on complex structured data, and a large number of indexes have been devised. Many of them adopt the SA scheme [5, 52] which splits the data objects into small sub-units and build inverted index on these sub-units. Different data types are broken down into different types of sub-units. Examples include words for documents [56, 43], q-grams for sequences [37, 61], binary branches for trees [64] and stars for graphs [63, 58, 60]. During query processing, the query object is also broken down into smaller sub-units and corresponding posting lists are scanned to identify data objects that share a lot of common sub-units with the query object. Sometimes, a verification step is necessary to compute the real distance (e.g. edit distance) between the candidates and the query object [61, 60, 64].

2.2 Parallelizing similarity search

Parallelism can be adopted to improve the throughput for similarity search. Different parallelization strategies must however be adopted for different data types. For example, Chen et al. [12] propose a novel data structure – successor table – for fast parallel sequence similarity search on an MPP computing model; whereas Tran et al. [57] propose a two-stage index to accelerate the sequence similarity search on both GPUs/many cores. There are also some proposed index structures on graphs and trees that can be parallelized [55, 63]. However, these indexes that are tailored for special data types cannot be easily extended to support similarity search for other data types.

There are also a few GPU-based methods for ANN search using LSH. Pan et al. [44] propose a searching method on the GPU using a bi-level LSH algorithm [45]. Lukac et al.[40] study the LSH method with multi-probe variant using GPUs. Both methods are specially designed for ANN search in the $l_{\mathfrak{p}}$ space. However GENIE can generally support LSH for ANN search under various similarity measures.

2.3 Data structures and indexes on the GPU

There are some works [16, 62, 4] about inverted index on the GPU to design specialized algorithms for accelerating some important operations on search engines. A two-level inverted-like index on the GPU is also studied for continuous time series search under the Dynamic Time Warping distance [70]. However, to the extent of our knowledge, there is no existing work on inverted index framework for generic similarity search on the GPU.

Tree-based data structures are also investigated to fully utilize the parallel capability of the GPU. Parallel accessing to the B-tree[69, 21, 7] or R-tree [41, 31, 66] index on the GPU memory is studied for efficiently handling query processing. Kim et al. propose an architecture sensitive layout of the index tree exploiting thread-level and data-level parallelism on both CPUs and GPUs [30]. In order to leverage the computational power as well as to overcome the current memory bottleneck of the GPU, some heterogeneous CPU-GPU systems for key-value store are also proposed [68, 23].

3. FRAMEWORK OVERVIEW

In this section, we give an overview of our system. We first present a conceptual model which formally defines the data and query in our system. Then we introduce the computational framework of our system.

3.1 Match-count model for inverted index

We first give several preliminary definitions about the data and query to facilitate the description of our model and index structure. Given a universe U, an **object** O_i contains a set of elements in U, i.e. $O_i = \{o_{i,1}, ..., o_{i,r}\} \subset U$. A set of such data objects forms a **data set** $DS = \{O_1, ...O_n\}$. A **query** Q_i is a set of items $\{q_{i,1}, ..., q_{i,s}\}$, where each item $q_{i,j}$ is a set of elements from U, i.e. $q_{i,j} \subset U$ ($q_{i,j}$ is a subset of U). A **query set** is defined as $QS = \{Q_1, ..., Q_m\}$.

To understand the definitions of the query and the object, we give two examples instancing them as real data types. One example is the document data, where the universe U contains all possible words. In this case, the object O_i is a document comprising a set of words, while the query Q is a set of items where each item may contain one or more words. Another example is relational data, described as follows.

EXAMPLE 3.1. *Given a relational table, the universe U is a set of* ordered pairs (d, v) *where d indicates an attribute of this table and v is a value of this attribute. An $l-$dimensional relational tuple $p = (v_1, ..., v_l)$ can be represented as an object $O_i = \{(d_1, v_1), (d_2, v_2), ..., (d_l, v_l)\}$. A query on the relational table usually defines a set of ranges R=($[v_1^L, v_1^U]$, $[v_2^L, v_2^U]$,..., $[v_l^L, v_l^U]$). By our definition, the query can be represented as Q=$\{(d_1, [v_1^L, v_1^U])$,$(d_2, [v_2^L, v_2^U])$,..., $(d_l, [v_l^L, v_l^U])\}$, where $(d_i, [v_i^L, v_i^U])$ is an infinite set of ordered pairs (d_i, v) each comprised of a dimension d_i and a value $v \in [v_i^L, v_i^U]$. A simple example of queries and objects for a relational table is shown in Figure 1.*

Informally, given a query Q and an object O, the match-count model $MC(\cdot, \cdot)$ returns the number of elements $o_{i,j} \in O_i$ contained by at least one query item of Q. We give a formal definition of the match-count model as follows.

DEFINITION 3.1 (MATCH-COUNT MODEL). *Given a query* $Q = \{q_1, q_2, .., q_s\}$ *and an object* $O = \{o_1, ..., o_r\}$*, we map each query item* q_i *to a natural integer using* $C : (q_i, O) \rightarrow \mathbb{N}_0$*, where* $C(q_i, O)$ *returns the number of elements* $o_j \in O$ *contained by the item* q_i *(which is also a subset of* U*). Finally the output of the match-count model is the sum of the integers* $MC(Q, O) = \sum_{q_i \in Q} C(q_i, O)$.

We can rank all the objects in a data set with respect to the query Q according to the model $MC(\cdot, \cdot)$. After computing all the counts, we can output the top-k objects with the largest value of $MC(\cdot, \cdot)$.

3.2 Computational framework

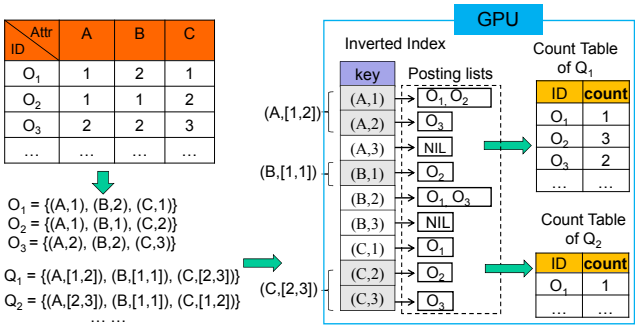


Figure 1: An illustration of the inverted index on a relational table. Multiple queries are processed, accessing the inverted index in parallel.

In this section, we introduce a computational framework for the match-count model. Essentially, the framework is based on an inverted index, which can fully utilize the GPU parallel computation capability in a fine-grained manner. We give a brief introduction of the computational architecture of the GPU in Appendix A.

Figure 1 shows an illustration of the high level inverted index on a relational table for GENIE. We first encode attributes and all possible values as ordered pairs. Continuous valued attributes are first discretized. Then we construct an inverted index where the *keyword* is just the encoded pair and the *posting list* comprises all objects having this keyword. Given a query, we can quickly map each query item to the corresponding keywords (ordered pairs). After that, by scanning the posting lists, we can calculate the match counts between the query and all objects.

In our GENIE system, a main component is a specially designed inverted index on the GPU to compute the match-count model for multiple queries, which is introduced in Section 4. We describe how to generalize our match-count model for searching on different data types and under different similarity measures in Section 5 and Section 6.

4. INVERTED INDEX ON THE GPU

In this section, we discuss the system design of GENIE. We first present the index structure and the data flow of GENIE. Then we present a novel data structure – Count Priority Queue (c-PQ for short), which is a priority queue-like structure on the GPU memory facilitating the search on the Count Table especially for top-k related queries.

4.1 Multiple queries on inverted index

The inverted index is resident in the global memory of the GPU. Figure 2 illustrates an overview of such an index structure. On the GPU, all the posting lists are stored in a large *List Array* in the GPU's global memory. There is also a *Position Array* which stores the starting position of each posting list for each keyword in the List Array. In addition, all the keywords (ordered pairs) are mapped to the Position Array on the GPU. For the multiple dimensional/relational data, where the ordered pair is (d_i, v_j) with d_i being dimension identifier and v_j being discretized value or category, we can use a simple function to encode the dimension (in high bits) and value (in low bits) into an integer, which is also the keyword's address of the Position Array. In this case, the total size of the Position Array is $O(dim * number\_of\_values)$. However, if an attribute of the data has many possible categories/values, while the existing objects only appear on a few of the categories/values, it is wasteful to use such an encoding method to build a big but almost empty Position Array. In this case, we maintain a hash structure to build a bijective map between the ordered pair and the Position Array.

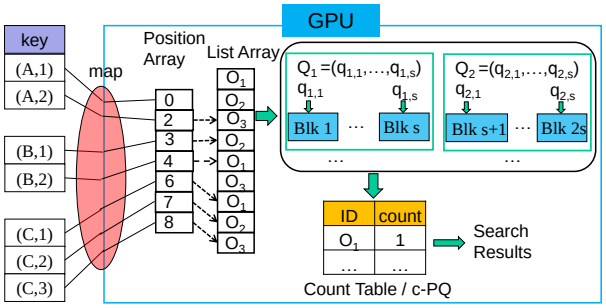


Figure 2: Overview of the inverted index and data flow on the GPU.

GENIE is designed to support multiple queries on the GPU. Figure 2 shows the processing of multiple queries. Each query has a set of items, which define particular ranges on some attributes (that is why we say each item is a subset of the universe U). After invoking a query, we use one block of the GPU to scan the corresponding posting lists for each query item, where each block has many threads (up to 1024) to access the posting lists in parallel. For a query $Q_i = \{q_{i,1}, q_{i,2}, ..., q_{i,s}\}$ with s query items, we invoke s blocks in total. If there are m queries, there will be about $m \cdot s$ blocks working on the GPU in parallel. During the processing, we update the *Count Table* which records the number of occurrences of the objects in the scanned posting lists. Therefore, the system is working in a fine-grained manner to process multiple queries which fully utilizes the parallel computational capability of the GPU.

In the inverted index, there may be some extreme long posting lists, which can become the bottleneck of our system. We also consider how to balance the workload for each block by breaking long posting lists into short sub-lists. We refer readers to Appendix B for more details about the load balance.

4.2 Count Priority Queue

The major problem of the Count Table is its large space cost, which requires allocating an integer to store the count for each object for each query. Taking a dataset with 10M

points as an example, if we want to submit a batch of one thousand queries, the required space of the Count Table is about 40 GB (by allocating one integer for count value, the size is $1k(queries) \times 10M(points) \times 4(bytes) = 40GB$), which exceeds the memory limit of the current available GPUs.

Another problem is how to select the top-k count objects from the Count Table. It is desirable to use a priority queue for this problem. However, the parallel priority queue cannot run efficiently on current GPUs' architectures [65, 71, 45, 22]. As far as known, the most efficient priority queue solution on the GPU for this problem is to employ a k-selection algorithm using an array [1]. We refer readers to Appendix C for a brief explanation of this priority queue solution. Nevertheless, such solution requires storing object id explicitly. Taking the example of 10M points again, if we want to submit a batch of one thousand queries, the required space to store the object id of the method in [1] is also 40 GB (by allocating one integer for object id, the size is $1k(queries) \times 10M(points) \times 4(bytes) = 40GB$). Such a high space consumption limits the number of queries being processed in a batch on the GPU.

In this section, we propose a novel data structure, called Count Priority Queue (c-PQ for short) to replace the Count Table, which aims to improve the efficiency and throughput of GENIE. c-PQ has three strong points. First, c-PQ does not require explicitly storing id for most of the objects which can significantly reduce the space cost. Second, c-PQ can return the top-k results within a very small cost. Third, c-PQ makes it possible to adopt the bitmap to further compress the space requirement of GENIE.

The key idea of c-PQ is to use a two-level data structure to store the count results, and use a device to schedule the data allocation between levels. Figure 3 shows the three main components of c-PQ. In the lower level, we create a *Bitmap Counter* which allocates several bits (up to 32 bits) for each objects whose id is corresponding to the beginning address of the bits. Thus, the object id does not explicitly stored in the Bitmap Counter. In the upper level, there is a *Hash Table* whose entry is a pair of object id and its count value. Then, a pivotal device, called *Gate*, determines which id-value pair in the Bitmap Counter will be inserted into the Hash Table. We will prove an important property of c-PQ later that only a few objects can pass the Gate to the Hash Table, while all remaining objects in the Bitmap Counter cannot be top-k objects and thus can be safely abandoned. Moreover, another extraordinary property of c-PQ is that, after updating, the Gate can return a threshold value for the top-k result, and we only need to scan the Hash Table once with the threshold to select all the top-k count objects. In next section, we will give a detailed introduction about the structure and mechanism of c-PQ, and prove all the claims in Theorem 4.1.

4.2.1 The structure and mechanism of c-PQ

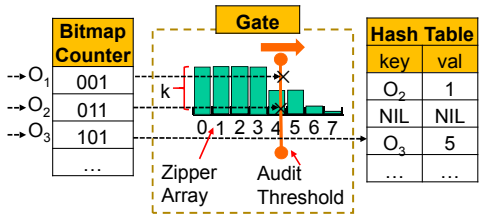


Figure 3: An illustration of the c-PQ.

Beside reducing the space cost by implicitly storing the object id, the Bitmap Counter can also compress the space compared with a native Count Table. If the maximum count is bounded, we only need to allocate a smaller number of bits (instead of 32 bits) which are enough to encode the maximum count for each object. In most of the cases, the maximum count is bounded, for example, the dimensions of an object in l_p normal space is limited instead of infinity.

The Gate restricts objects from the Bitmap Counter going to the Hash Table. The objective is that, after scanning the posting lists, only objects in the Hash Table can be the potential candidates of the top-k results. For this purpose, the Gate has a *ZipperArray* whose size is equal to the maximum value of the count. *ZipperArray[i]* records the number of objects whose count have reached i. The Gate also maintains a threshold called *AuditThreshold* which records the minimum index of ZipperArray whose value is smaller than k (i.e. $ZipperArray[AuditThreshold] < k$ and $ZipperArray[AuditThreshold - 1] \geq k$).

We present the update process of c-PQ as well as the running mechanism of the Gate in Algorithm 1. Figure 3 also gives an illustration of the updating process. When the count of an object is updated in the Bitmap Counter, we immediately check whether the object's count is larger than the *AuditThreshold* (line 3). If it is (like O_3 whose count is 5 in Figure 3 and the *AuditThreshold* is 4), we will insert an entry into the Hash Table whose key is the object id and whose value is the object's count. If the object id is already in the Hash Table, we will replace the old entry with the new one. Meanwhile, we will update the *ZipperArray* (line 5 of Algorithm 1). If *ZipperArray[AuditThreshold]* is larger than k, we also increase the *AuditThreshold* by one unit (line 7 of Algorithm 1).

Algorithm 1: Update on the Count Priority Queue

```
   // After scanning object O_i in the inverted index
1  val_i = BitmapCounter[O_i] + 1
2  BitmapCounter[O_i] = val_i
3  if val_i ≥ AuditThreshold then
4      Put entry (O_i, val_i) into the Hash Table
5      ZipperArray[val_i] += 1
6      while ZipperArray[AuditThreshold] ≥ k do
7          AuditThreshold += 1
```

Now we use Theorem 4.1 to elaborate the properties of c-PQ mentioned in the previous section.

THEOREM 4.1. *After finishing the updating process of c-PQ (which happens at the same time with finishing scanning the inverted index), we claim that the top-k candidates are only stored in the Hash Table, and the number of candidate objects in the Hash Table is $O(k * AuditThrehsold)$. Suppose the match count of the k-th object O_k of a query Q is $MC_k = MC(Q, O_k)$, then we have $MC_k = AuditThrehsold - 1$.*

PROOF. See Appendix F.1. □

With the help of Theorem 4.1, we can select the top-k objects by scanning the Hash Table and selecting objects with match count greater than $(AuditThrehsold - 1)$ only. If there are multiple objects with match count equal to $(AuditThrehsold - 1)$, we break ties randomly.

4.2.2 Hash Table with modified Robin Hood Scheme

In this section, we briefly discuss the design of Hash Table in c-PQ. We propose a modified Robin Hood Scheme to implement a hash table on the GPU which is different from existing work [18]. According to Theorem 4.1, the size of the Hash Table can be set as $O(k*max\_count\_value)$. More discussion about the Robin Hood Scheme and the Hash Table on the GPU can be found in Appendix D.

The vital insight to improve the efficiency of the Hash Table with Robin Hood Scheme in the c-PQ is that all entries with values smaller than $(AuditThrehsold - 1)$ in the Hash Table cannot be top-k candidates (see Theorem 4.1). If the value of an entry is smaller than $(AuditThrehsold - 1)$, we can directly overwrite the entry regardless of hashing confliction. In this way, we can significantly reduce the probe times of insertion operations of the Hash Table, since many inserted keys become expired with the monotonous increase of *AuditThreshold*.

5. GENERIC ANN SEARCH

In this section and next section, we discuss how to generalize our match-count model for similarity search of various data types. Here we first show that GENIE (with match-count model) can support the ANN search for any similarity measure which has a generic LSH scheme. For complex data types with no LSH transformation, we also showcase the SA scheme of GENIE for similarity search in Section 6.

5.1 ANN search by generic similarity measure

Approximate Nearest Neighbor (ANN) search has been extensively studied. A popular definition of the ANN search is the *c−approximate nearest neighbor* (*c*-ANN) search [26], which is defined as: to find a point p so that $sim(p, q) \leq c \cdot sim(p^*, q)$ with high probability where p^* is the true nearest neighbor. Here $sim(\cdot, \cdot)$ is a function that maps a pair of points to a number in $[0, 1]$ where $sim(p, q) = 1$ means p and q are identical. LSH is one of the most popular solutions for the ANN search problem [26, 11, 59].

The problem to integrate existing LSH methods into GENIE is to present a theoretical bound analysis for LSH under match-count model. For this purpose, we first propose a revised definition of the ANN search, called Tolerance-Approximate Nearest Neighbor search (τ-ANN), which holds the same spirit of the definition of *c*-ANN:

DEFINITION 5.1 (TOLERANCE-ANN, τ-ANN). *Given a set of n points $P = \{p_1, p_2, .., p_n\}$ in a space S under a similarity measure $sim(p_i, q)$, the Tolerance-Approximate Nearest Neighbor (τ-ANN) search returns a point p such that $|sim(p, q) - sim(p^*, q)| \leq \tau$ with high probability where p^* is the true nearest neighbor.*

Note that some existing works, like [50], have used a concept similar to Definition 5.1 though without explicit definition.

According to the definition in [11], a hashing function $h(\cdot)$ is said to be locality sensitive if it satisfies:

$$Pr[h(p) = h(q)] = sim(p, q) \tag{1}$$

which means the collision probability is equal to the similarity measure. Another definition of LSH is formulated that the collision probability is larger than a threshold when two points are close enough. We postpone the discussion of this definition to Section 5.2. Next, we introduce how to integrate LSH into our GENIE system, followed by an estimation of error bound on the GENIE. A discussion about the GENIE from the view of maximum likelihood estimation (MLE) can be found in Appendix E.

5.1.1 Building index for LSH and ANN search

We can use the indexing method shown in Figure 1 to build an inverted index for LSH. In this case, we treat each hash function as an attribute, and the hash signature as the value for each data point. The keyword in the inverted index for point p under hash function $h_i(\cdot)$ is a pair (h_i, v) and the posting list of pair (h_i, v) is a set of points whose hash value by $h_i(\cdot)$ is v (i.e. $h_i(p) = v$).

A possible problem is that the hash signature of LSH functions may have a huge number of possible values with acceptable error by configuring the LSH parameters. For example, the signature of the Random Binning Hashing function introduced later can be thousands of bits by setting good parameters for search error. Meanwhile, it is not reasonable to discretize the hash signature into a set of buckets, since the hash signature of two points in the same discretized bucket would have different similarity, which will validate the definition of LSH in Eqn. 1.

To reduce the number of possible values introduced by LSH, we propose a random re-hashing mechanism. After obtaining the LSH signature $h_i(\cdot)$, we further randomly project the signatures into a set of buckets with a random projection function $r_i(\cdot)$. Figure 4 shows an example of the re-hashing mechanism. We can convert a point to an object of our match-count model by the transformation: $O_i = [r_1(h_1(p_i)), r_2(h_2(p_i)), ..., r_m(h_m(p_i))]$ where $h_j(\cdot)$ is an LSH function and $r_j(\cdot)$ is a random projection function. Note that the re-hashing mechanism is generic: it can be applied to any LSH hashing signature. Re-hashing is not necessary if the signature space of selected LSH can be con
gured small enough.

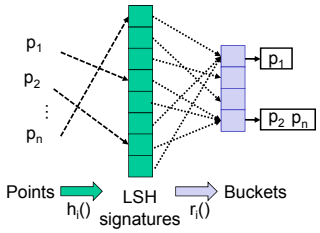


Figure 4: Re-hashing mechanism. The hashed object is $O_i = [r_1(h_1(p_i)), r_2(h_2(p_i)), ..., r_m(h_m(p_i))]$ where $h(\cdot)$ is a LSH function and $r(\cdot)$ is a random projection function.

After building the inverted index, the ANN search can be conveniently supported by GENIE. Given a query point q, we also convert q with the same transformation process shown in Figure 4, i.e. $Q = [r_1(h_1(q)), r_2(h_2(q)), ..., r_m(h_m(q))]$. Then the top result returned by GENIE is just the τ-ANN search result, which is proved in the following section.

5.1.2 Error bound and τ-ANN

In this section, we prove that the top return of GENIE for a query q is the τ-ANN of q. Given a point p and a query q with a set of LSH functions $\mathbb{H} = \{h_1, h_2, ..., h_m\}$ defined by Eqn. 1, suppose there are c functions in $\mathbb{H}$ satisfying $h_i(p) = h_i(q)$ (where c is just the return of match-count model). We first prove that with properly configuring the number of hash

functions in GENIE (which is denoted as m), the return of match-count model on GENIE can be probabilistically bounded w.r.t the similarity between p and q, i.e. $|c/m - sim(p,q)| < \epsilon$ with high probability. This error bound is clarified in Theorem 5.1.

THEOREM 5.1. *Given a similarity measure $sim(\cdot,\cdot)$, an LSH family $h(\cdot)$ satisfied Eqn. 1, we can get a new hash function $f(x) = r(h(x))$, where $r(\cdot)$ is a random projection function from LSH signature to a domain $R : U \to [0, D)$.*

For a set of hash functions $f_i(\cdot) = r_i(h_i(\cdot)), 1 \le i \le m$ with $m = 2\frac{ln(3/\delta)}{\epsilon^2}$, we can convert a point p and a query q to an object and a query of the match-count model, which are $O_p = [f_1(p), f_2(p), ..., f_m(p)]$ and $Q_q = [f_1(q), f_2(q), ..., f_m(q)]$, then we have $|MC(O_p, Q_q)/m - sim(p,q)| < \epsilon + 1/D$ with probability at least $1 - \delta$.

PROOF. See Appendix F.2. □

Now we introduce an important theorem which claims that, given a query point q and proper configuration of m stated in Theorem 5.1, the top result returned by GENIE is just the τ-ANN of q.

THEOREM 5.2. *Given a query q and a set of points $P = \{p_1, p_2, .., p_n\}$, we can convert them to the objects of our match-count model by transformation $O_{p_i} = [r_1(h_1(p_i)), r_2(h_2(p_i)), ..., r_m(h_m(p_i))]$ (as shown in Figure 4) which satisfies $|MC(O_{p_i}, Q_q)/m - sim(p_i, q)| \le \epsilon$ with the probability at least $1 - \delta$. Suppose the true NN of q is p^*, and the top result based on the match-count model is p, then we have $sim(p^*, q) - sim(p, q) \le 2\epsilon$ with probability at least $1 - 2\delta$.*

PROOF. See Appendix F.3. □

5.1.2.1 Number of hash functions in practice.

Theorem 5.1 provides a rule to set the number of LSH functions where the number of hash functions is proportional to the inverse of squared error $O(\frac{1}{\epsilon^2})$ which may be very large. It is NOT a problem for GENIE to support such a number of hash functions since the GPU is a parallel architecture suitable for the massive quantity of relatively simple tasks. The question however is that: Do we really need such a large number of hash functions in practical applications?

Before exploiting this, we first explain that the collision probability of a hash function $f_i(\cdot)$ can be approximated with the collision probability of an LSH function $h_i(\cdot)$ if D is large enough. The collision probability of $f_i(\cdot)$ can be divided into two parts: collisions caused by $h_i(\cdot)$ and collisions caused by $r_i(\cdot)$, which can be expressed as:

$$Pr[f_i(p) = f_i(q)] = Pr[r_i(h_i(p)) = r_i(h_i(q))] \quad (2)$$
$$\le Pr[h_i(p) = h_i(q)] + Pr[r_i(h_i(p)) = r_i(h_i(q))] \quad (3)$$
$$= s + 1/D \quad (4)$$

where $s = sim(p,q)$. Thus, we have $s \le Pr[f_i(p) = f_i(q)] \le s + 1/D$. Suppose $r(\cdot)$ can re-hash $h_i(\cdot)$ into a very large domain $[0, D)$, we can claim that $Pr[f_i(p) = f_i(q)] \approx s$. For simplicity, let us denote $c = MC(Q_q, O_p)$. An estimation of s by maximum likelihood estimation (MLE) can be [50]:

$$s = MC(Q_q, O_p)/m = c/m \quad (5)$$

Eqn. 5 can be further justified by the following equation:

$$Pr[|\frac{c}{m} - s| \le \epsilon] = Pr[(s-\epsilon) * m \le c \le (s+\epsilon) * m] \quad (6)$$
$$= \sum_{c=\lfloor (s-\epsilon)m \rfloor}^{\lceil (s+\epsilon)m \rceil} \binom{m}{c} s^c (1-s)^{m-c} \quad (7)$$

The problem of Eqn. 7 is that the probability of error bound depends on the similarity measure $s = sim(p,q)$ [50]. Therefore, there is no closed-form expression for such error bound.

Nevertheless, Eqn. 7 provides a practical solution to estimate a tighter error bound of the match-count model different from Theorem 5.1. If we fixed ϵ and δ, for any given similarity measure s, we can infer the number of required hash functions m subject to the constraint $Pr[|c/m - s| \le \epsilon] \ge 1 - \delta$ according to Eqn. 7. Figure 5 visualizes the number of minimum required LSH functions for different similarity measure with respect to a fixed parameter $\epsilon = \delta = 0.06$ by this method. A similar figure has also been illustrated in [50]. As we can see from Figure 5, the largest required number of hash functions, being m=237, appears at $s = 0.5$, which is much smaller than the one estimated by Theorem 5.1 (which is $m = 2\frac{ln(3/\delta)}{\epsilon^2} = 2174$). We should note that the result shown in Figure 5 is data independent. Thus, instead of using Theorem 5.1, we can effectively estimate the actually required number of LSH functions using the simulation result based on Eqn. 7 (like Figure 5).

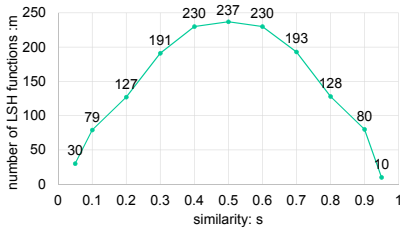


Figure 5: Similarity (s) v.s. the number of minimum required LSH functions (m) with constraint $Pr[|c/m - s| \le \epsilon] \ge 1 - \delta$ where $\epsilon = \delta = 0.06$. By Theorem 5.1, $m = 2\frac{ln(3/\delta)}{\epsilon^2} = 2174$.

5.1.3 Case study: τ-ANN in Laplacian kernel space

A fascinating property of GENIE is that any similarity measures associated with an LSH family defined by Eqn. 1 can be supported by GENIE for τ-ANN search.

In this section, we take the ANN search on a shift-invariant kernel space as a case study, which has important applications for machine learning and computer vision. The authors in [47] propose an LSH family, called Random Binning Hashing (RBH), for Laplacian kernel $k(p,q) = exp(- \parallel p - q \parallel_1 /\sigma)$. Though this method is well-known for dimension reduction, as far as we know, it has not been applied to ANN search. One possible reason is that this method requires a huge hash signature space, where the number of bits required is a multiple of the number of dimensions of points. A brief introduction to RBH is in Appendix G.

In experiment, we demonstrate that GENIE can support ANN search in Laplacian kernel space based on RBH. To reduce the hash signature space, we use the re-hashing mechanism to project each signature into a finite set of buckets.

5.2 ANN search in high dimensional space

For ANN search in high dimensional space, we usually resort to $(r_1, r_2, \rho_1, \rho_2)$-sensitive hashing function family which deserves our special discussion. A $(r_1, r_2, \rho_1, \rho_2)$-sensitive hashing function in $l_\mathfrak{p}$ norm space can be defined as [14]:

$$h(q) = \lfloor \frac{\mathbf{a}^T \cdot q + b}{w} \rfloor \quad (8)$$

where $\mathbf{a}$ is a random vector drawn from a $\mathfrak{p}$-stable distribution for $l_\mathfrak{p}$ distance function, and b is a random real number drawn uniformly from $[0, w)$. A more detailed discussion is in Appendix H.

In order to integrate such an LSH function into our proposed match-count model for ANN search, we have to find the relation between the collision probability and the $l_\mathfrak{p}$ distance. For this purpose, we justify the LSH function of Eqn. 8 by the following equation [14] (let $\Delta = \| p - q \|_\mathfrak{p}$):

$$\psi_\mathfrak{p}(\Delta) = Pr[h(p) = h(q)] = \int_0^w \frac{1}{\Delta}\phi_\mathfrak{p}(\frac{t}{\Delta})(1 - \frac{t}{w})dt \quad (9)$$

where $\phi_\mathfrak{p}(\Delta)$ denotes the probability distribution density function (pdf) of the absolute value of the $\mathfrak{p}$-stable distribution.

From Eqn. 9, we can infer that $\psi_\mathfrak{p}(\Delta)$ is a strictly monotonically decreasing function [3]: If p_1 is more nearby to q than p_2 in l_p space ($\Delta_1 = \| p_1 - q \|_\mathfrak{p} < \Delta_2 = \| p_2 - q \|_\mathfrak{p}$), then $\psi_\mathfrak{p}(\Delta_1)$ is higher than $\psi_\mathfrak{p}(\Delta_2)$. Therefore, we can say that $\psi_\mathfrak{p}(\Delta)$ defines a similarity measure between two points in l_p norm space, i.e.

$$sim_{l_\mathfrak{p}}(p, q) = \psi_\mathfrak{p}(\Delta) = \psi_\mathfrak{p}(\| p - q \|_\mathfrak{p}). \quad (10)$$

Recalling Theorem 5.1 and Theorem 5.2, the only requirement for the LSH function of GENIE is to satisfy Eqn. 1, which can be justified by Eqn. 9 for $(r_1, r_2, \rho_1, \rho_2)$-sensitive hashing function family. In other words, we can use the GENIE to do τ-ANN search under the similarity measure of Eqn. 10. Though the ANN search result is not measured by the $l_\mathfrak{p}$ norm distance, the returned results follow the same criterion to select the nearest neighbor since the similarity measure defined in Eqn.10 is closely related to the l_p norm distance.

A similar counting method has also been used for c-ANN search in [17] where a Collision Counting LSH (C2LSH) scheme is proposed for c-ANN search. Though our method has different theoretical perspective from C2LSH, the basic idea behind them is similar: the more collision functions between points, the more likely that they would be near each other. From this view, the C2LSH can corroborate the effectiveness of the τ-ANN search of our method.

6. SEARCHING ON ORIGINAL DATA

GENIE also provides a choice of adopting the "Shotgun and Assembly" (SA) scheme for similarity search. Given a dataset, we split each object into small units. Then we build inverted index on such dataset where each unique unit is a keyword, and the corresponding posting list is a list of objects containing this unique unit. When a query comes, it is also broken down as a set of such small units. After that, GENIE can effectively calculate the number of common units between the query object and data objects.

The return of the match-count model can either be considered as a similarity measure (such as document search where the count is just the inner product between the space vector of documents) or be considered as a lower bound of a distance (e.g. edit distance) to filter candidates [61, 60, 64]. In the following sections, we will showcase how to perform similarity search on sequence data, short document data and relational data using GENIE.

6.1 Searching on sequence data

In this section, we expound how to use GENIE to support the similarity search by SA scheme with an example of sequence similarity search under edit distance. The general process is to first chop the data sequences and query sequence as n-gram with sliding windows, and then to build inverted index on the GPU with the n-gram as keyword.

6.1.1 Shotgun: decomposition and index

Given a sequence S and an integer n, an n-gram is a subsequence s of S with length n. In this step, we will decompose the sequence S into a set of n-gram with a sliding window of length n. Since the same n-gram may appear multiple times in a sequence, we introduce the *ordered n-gram*, which is a pair of (n-gram,i) where i denotes the i-th same n-gram in the sequence. Therefore, we finally decompose the sequence S into a set of ordered n-gram $G(S)$. In GENIE, for each sequence we build inverted index on the GPU with treating the ordered n-gram as keyword and putting sequence id in the posting list.

EXAMPLE 6.1. *For a sequence $S = \{aabaab\}$, the set of* ordered 3-grams *of S is $G(S)\{(aab, 0), (aba, 0), (baa, 0), (aab, 1)\}$ where $(aab, 0)$ denotes the first subsequence aab in S, and $(aab, 1)$ denotes the second subsequence aab in S.*

6.1.2 Assembly: combination and verification

After building index, we will issue queries with resorting to GENIE. For a query sequence Q, we also decompose it into a set of *ordered n-grams* using sliding windows, i.e. $G(Q)$. Then we can follow the routine of GENIE to retrieve candidates with top-k large count in the index. Before explaining this method, we first propose the following lemma:

LEMMA 6.1. *Suppose the same n-gram s_n^i appear c_s^i times in sequence S and c_q^i times in sequence Q, then the return of the match-count model is $MC(G(S), G(Q)) = \sum_{s_n^i} min\{c_s^i, c_q^i\}$.*

With respect to the edit distance, the return of the match-count model has the following theorem [54].

THEOREM 6.1. *If the edit distance between S and Q is τ, then the return of the match-count model has $MC(G(S), G(Q)) \geq \max\{|Q|, |S|\} - n + 1 - \tau * n$. [54]*

According to Theorem 6.1, we can use the return of the match-count model as an indicator for selecting the candidates for the query sequence. Our strategy is to retrieve $\mathbf{K}$ candidates from GENIE according to match count with setting a large $\mathbf{K}(\mathbf{K} >> k)$. Then we can verify the edit distance between these $\mathbf{K}$ candidates and the query sequence to find k data sequences nearest to Q under edit distance.

With this method, we can know whether the real top-k sequences are correctly returned by GENIE, though we cannot guarantee that the returned top-k candidates are the real top-k data sequence for all queries. In other words, if we retrieve $\mathbf{K}$ candidates according to count from GENIE and the k-th most similar sequence under edit distance among these $\mathbf{K}$ candidates to Q is $S^{\mathbf{k}'}$, we can know whether $S^{\mathbf{k}'}$ is

the real k-th most similar sequence of Q. Formally, we have the following theorem.

THEOREM 6.2. *For the $\mathbf{K}$-th candidates $S^{\mathbf{K}}$ returned by GENIE according to count, suppose the match count between $S^{\mathbf{K}}$ and query Q is $c_{\mathbf{K}} = MC(G(S^{\mathbf{K}}), G(Q))$. Among the $\mathbf{K}$ candidates, after verification, the edit distance between k-th most similar sequence and Q is $\tau_{\mathbf{k}'} = ed(Q, S^{\mathbf{k}'})$. If $c_{\mathbf{K}} < |Q| - n + 1 - \tau_{\mathbf{k}'} * n$, then the real top-k results is correctly returned by GENIE.*

A possible solution for sequence similarity search is to repeat search process by GENIE, until the condition in Lemma 6.2 is satisfied. In the worst case it may need to scan the whole data set before retrieving the real top-k sequences under edit distance, if the bound in Theorem 6.1 cannot work well[4]. However, as we shown in our experiment, it is good enough for near edit distance similarity search with one round of the search process in some applications, such as sequence error correction.

6.2 Searching on short document data

In this application, both the query document and the object document are broken down into "words". We build an inverted index with GENIE where the keyword is a "word" from the data document, and the posting list of a "word" is a list of document ids.

We can explain the result returned by GENIE on short document data by the document vector space model. Documents can be represented by a binary vector space model where each word represents a separate dimension in the vector. If a word occurs in the document, its value in the vector is one, otherwise its value is zero. The output of the match-count model, which is the number of co-occurred words in both the query and the object, is just the *inner product* between the binary sparse vector of the query document and the one of the object document. In our experiment, we show an application of similarity search on a tweet dataset.

6.3 Searching on relational data

GENIE can also be used to support query on the relational data under the match-count model. In Figure 1, we have shown how to build an inverted index for relational tuples. A query on the relational table is a set of specific ranges on attributes of the relational table.

The top-k result returned by GENIE on relational tables can be considered a special case of the traditional top-k selection query. The top-k selection query selects the k tuples in a relational table with the largest predefined ranking score function $F(\cdot)$ (SQL *ORDER BY* $F(\cdot)$) [24]. In GENIE, we use a special ranking score function defined by the match-count model, which is especially useful for tables having both categorical and numerical attributes.

7. EXPERIMENTS

7.1 Settings

7.1.1 Datasets

We use five real-life datasets to evaluate our system. Each dataset corresponds to one similarity measure respectively introduced in Section 5 and Section 6.

[4]At first, we should make sure the bound of Theorem 6.1 is positive with the estimated edit distance.

[**OCR**][5] This is a dataset for optical character recognition [53]. It contains 3.5M data points and each point has 1156 dimensions. We randomly select 10K points from the dataset as query/test set (and remove them from the dataset). We use RBH to generate the LSH signature, which is further re-hashed into an integer domain of [0,8192) (see Section 5.1.3). The prediction performance is used to evaluate the quality of ANN search.

[**SIFT**][6] This dataset [28] contains 4.5M SIFT features [39] extracted from 1,491 photos, and each feature is a 128-dimensional point. We randomly select 10K features as our query set and remove these features from the dataset. We select the hash functions from the E2LSH family [14, 3] and each function transforms a feature into 67 buckets. The setting of the bucket width follows the routine in [14]. We use this dataset to evaluate the ANN search in high dimensional space as discussed in Section 5.2.

[**DBLP**][7] This dataset is obtained by extracting article title from DBLP website. The total number of sequences is 5.0M. We randomly choose 10K sequences as test data, and then modify 20% of the characters of the sequences. This dataset is to serve the experiment of sequence similarity search in Section 6.1. Specially, for the parameters discussed in Section 6.1, we set $\mathbf{K} = 500$ and $k = 1$. Note that we only use one round of the search process without further iteration, thus, we can know that some queries may not find the true top-k candidates according to Theorem 6.2. More discussion about this experiment can be found in Section 7.4.2.

[**Tweets**][8] This dataset has 6.8M tweets. We remove stop words from the tweets. The dataset is crawled by our collaborators from Twitter for three months by keeping the tweets containing a set of keywords[9]. We reserve 10K tweets as a query set for the experiment. It is used to study the short document similarity search (see Section 6.2).

[**Adult**][10] This dataset has census information [38]. It contains 4.9K rows with 14 attributes (mixed of numerical and categorical ones). For numerical data, we discretize all value into 1024 intervals of equal width. We further duplicate every row 20 times. Thus, there are 0.98M instances after duplication. We select 10K tuples as queries. For numerical attributes, the query item range is defined as $[discretized\_value - 50, discretized\_value + 50]$. We use this dataset to study the selection from relational data (see Section 6.3).

7.1.2 Competitors

We use the following competitors as baselines to evaluate the performance of GENIE.

[**GPU-LSH**] We use GPU-LSH [45] as a competitor of GENIE for ANN search in high dimensional space. Furthermore, since there is no GPU-based LSH method for ANN search in Laplacian kernel space, we still use GPU-LSH method as a competitor for ANN search of GENIE using

[5]http://largescale.ml.tu-berlin.de/instructions/

[6]http://lear.inrialpes.fr/~jegou/data.php

[7]http://dblp.uni-trier.de/xml/

[8]https://dev.twitter.com/rest/public

[9]The keywords include "Singapore", "City", "food joint" and "restaurant", etc. It is crawled for a research project.

[10]http://archive.ics.uci.edu/ml/datasets/Adult

generic similarity measure. We adjust the configuration parameters of GPU-LSH to make sure the ANN search results of GPU-LSH have similar quality with GENIE, which is discussed in Section 7.4.1. We only use 1M data points for GPU-LSH on OCR dataset since it cannot work more data points on this dataset.

[**GPU-SPQ**] We also implemented a priority queue-like method on GPU as a competitor. We first scan the whole dataset to compute match-count value between queries and all points in the dataset and store these computed results in an array, then we use a GPU-based fast k-selection [1] method to extract the top-k candidates from the array for each query. We name this top-k calculation method as SPQ (which denotes GPU fast k-selection from an array as a priority queue). We give a brief introduction to SPQ in Appendix C. Note that for ANN search, we scan on the LSH signatures (not original data) since our objective is to verify the effectiveness of the index.

[**CPU-Idx**] We also implemented an inverted index on the CPU memory. While scanning the inverted index in memory, we use an array to record the match-count value for each object. Then we use a partial quick selection function (with $\Theta(n + klogn)$ worst-case performance) in C++ STL to get the k largest-count candidate objects.

[**AppGram**] This is one of the state-of-the-art methods for sequence similarity search under edit distance on the CPU [61]. We use AppGram as a baseline for comparing the running time of GENIE for sequence similarity search. Note that AppGram and GENIE is not completely comparable, since the AppGram try its best to find the true kNNs, but GENIE only does one round search process without further iteration in the experiment, thus some true kNNs of queries may be missed (though we know which queries do not have true top-k, and another search process can be issued to explore the true kNNs). We give more discussion about this in Section 7.4.2.

[**GEN-SPQ**] This is a variant of GENIE but using SPQ instead of c-PQ. We still build inverted index on the GPU for each dataset. However, instead of using c-PQ structure introduced in Section 4.2, we use SPQ (which is the same with the one for GPU-SPQ) to extract candidates from the Count Table.

7.1.3 Environment

We conducted the experiment on a CPU-GPU platform. The GPU is NVIDIA GeForce GTX TITAN X with 12 GB memory. All GPU codes were implemented with CUDA 7. Other programs were implemented in C++ on CentOS 6.5 server (with 64 GB RAM). The CPU is Intel Core i7-3820.

If not otherwise specified, we set $k = 100$ and set the submitted query number per batch to the GPU as 1024. All the reported results are the average of running results of ten times. By default, we do not enable the load balance function since it is not necessary when the query number is too large for one batch process (see Section B.1). For ANN search (on OCR data and SIFT data), we use the method introduced in Section 5.1.2 to determine the number of LSH hash functions with setting $\epsilon = \delta = 0.06$, therefore the number of hash functions is $m = 237$.

7.2 Efficiency of GENIE

7.2.1 Search time for multiple queries

Table 1: Time profiling of different stages of GENIE for 1024 queries (the unit of time is *second*).

Stage		OCR	SIFT	DBLP	Tweets	Adult
Index build		81.39	47.73	12.53	12.10	1.06
Index transfer		0.53	0.34	0.27	0.088	0.011
Query	transfer	0.015	0.018	0.024	0.0004	0.0004
	match	2.60	7.04	3.30	1.19	1.82
	select	0.004	0.003	14.00*	0.003	0.009

*This includes verification time which is the major cost.

In this section, we compare the running time among GENIE and its competitors. Note that we do not include the index building time for all the competitors since these tasks can be done by offline processes. The index building time of GENIE will be discussed in Section 7.2.2.

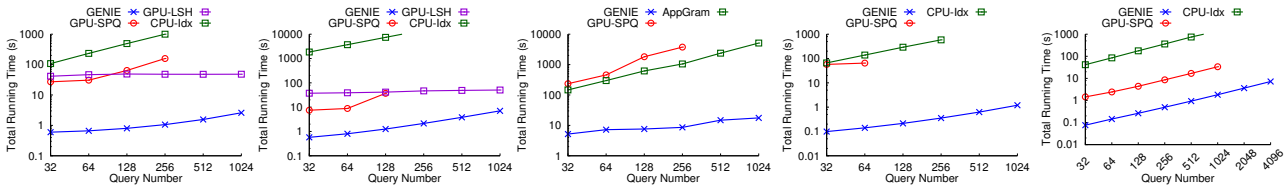
We show the total running time with respect to different numbers of queries in Figure 6 (y-axis is log-scaled). We do not show the running time for GPU-SPQ if the query number is too large to be supported in one batch. The running time of CPU-Idx is also hidden if it is too large to be shown in the figure. Our method also outperforms GPU-SPQ by more than one order of magnitude. Furthermore, GPU-SPQ can only run less than 256 queries in parallel (except for Adult dataset) for one batch process, but GENIE can support more than 1000 queries in parallel.

As we can see from Figure 6, GENIE can also achieve better performance than GPU-LSH. The running time of GPU-LSH is relatively stable with varying numbers of queries. This is because GPU-LSH uses one thread to process one query, thus, GPU-LSH achieves its best performance when there are 1024 queries (which is the maximum number of threads per block on the GPU). This also clarifies why GPU-LSH is even worse than the GPU-SPQ method when the query number is small since parts of the computational capability are wasted in this case. We can see that even when GPU-LSH comes into the full load operation, GENIE can still achieve better running time than GPU-LSH. Note that we only use 1M data points for GPU-LSH on OCR dataset. GENIE can achieve more than two orders of magnitude over GPU-SPQ and AppGram for sequence search. More experiments about the average query time can be found in [**?**].

Figure 7 conveys the running time of GENIE and its competitors with varying numbers of data points from each dataset. Since most of the competitors cannot run 1024 queries for one batch, we fix the query number as 512 in this figure. The running time of GENIE is gradually increased with the growth of data size, except the one on DBLP which is stable because the running time is dominated by edit distance verification cost (which is determined by parameter **K**). Nevertheless, the running time of GPU-LSH is relative stable on all datasets with respect to the data size. The possible reason is that GPU-LSH uses many LSH hash tables and LSH hash functions to break the data points into short blocks, therefore, the time for accessing the LSH index on the GPU become the main part of query processing.

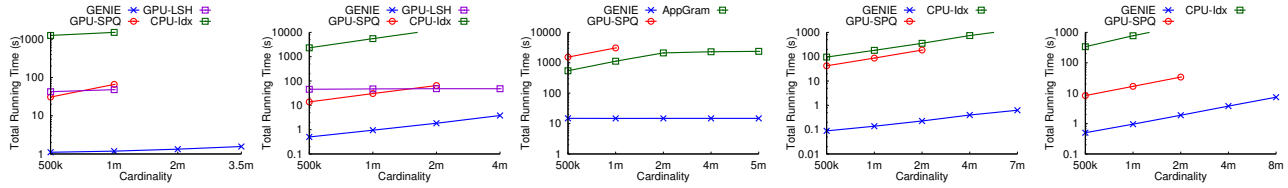
7.2.2 Time profiling

Table 1 shows the time cost of different stages of GENIE. The "Index-build" represents the running time to build the inverted index on the CPU. This is a one-time cost, and we do not count it into the query time. The "Index-transfer"



(a) OCR (b) SIFT (c) DBLP (d) Tweets (e) Adult

Figure 6: Total running time for multiple queries.



(a) OCR (b) SIFT (c) DBLP (d) Tweets (e) Adult

Figure 7: Varying data size for multiple queries (The query number is 512).

displays the time cost to transfer the inverted index from the CPU to the GPU. The rows of "Query" display the time for similarity search with 1024 queries per batch. The "Query-transfer" is the time cost to transfer queries and other related information from the CPU to the GPU. And the "Query-select" contains the time for selecting the candidates from c-PQ and sending back the candidates to the CPU memory (For DBLP data, we also invoke a verification process by computing the true edit distance between the query and candidates in this stage, so the time cost of Query-select for DBLP is much larger). The "Query-match" is just the time cost for scanning the inverted index, which dominates the cost for similarity search of multiple queries. Since the cost for transforming data is quite small, it is reasonable to use the GPU to accelerate such expensive index processing operation. This justifies the rationality for developing an inverted index on the GPU.

7.3 Effectiveness of c-PQ

Figure 8 demonstrates the effectiveness of c-PQ in GENIE. c-PQ can significantly reduce the memory requirement per query for GENIE. This is reflected in Figure 8 that the deployment of c-PQ results in more than 4-fold increase of the maximum number of queries for parallel query process on the GPU within one batch for every datasets. Besides, c-PQ can also reduce the running time by avoiding selecting the candidates from a large Count Table for multiple queries. As we can see from Figure 8, when the number of queries is the same, with the help of c-PQ, the running time of GENIE is also decreased.

7.4 Effectiveness of GENIE

In this section, we evaluate the effectiveness of GENIE under both LSH scheme and SA scheme, which are discussed in Section 7.4.1 and Section 7.4.2 respectively.

7.4.1 ANN Search of GENIE

In this section, we discuss the quality of the ANN search of GENIE as well as the parameter setting for GPU-LSH. An evaluation metric for the approximate kNN search in high dimensional space is *approximation ratio*, which is defined as how many times farther a reported neighbor is compared to the real nearest neighbor. In the experiment evaluation for

Table 2: Prediction result of OCR data by 1NN

method	precision	recall	F1-score	accuracy
GENIE	0.8446	0.8348	0.8356	0.8374
GPU-LSH	0.7875	0.7730	0.7738	0.7783

running time, we set the parameters of GPU-LSH and GENIE to ensure that they have similar approximation ratio. A formal definition of approximation ratio and the parameter setting method can found in Appendix I.2.

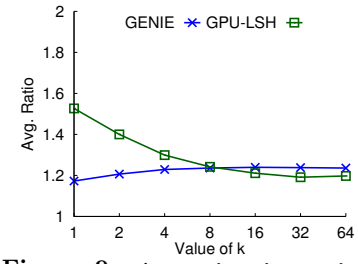
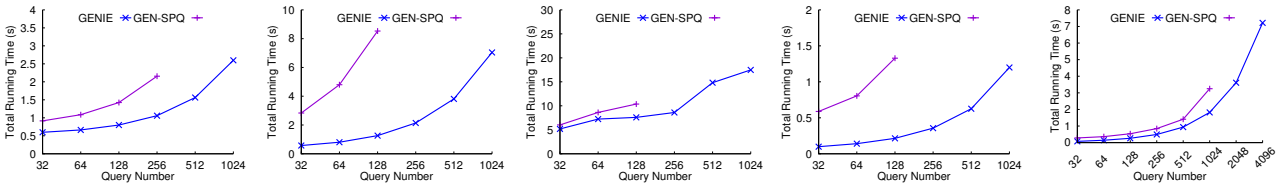


Figure 9: Approximation ratio v.s. value of k on SIFT data

Figure 9 shows the approximation ratio of GPU-LSH and GENIE with varying the value of k. From Figure 9, we can see that GENIE has stable approximation ratio with varying the k; whereas, GPU-LSH has large approximation ratio when k is small and converges with the one of GENIE when k increases. One possible reason of this phenomenon is that GPU-LSH needs to verify a few set of candidates to determine the final returned kNNs while the number of verified candidates is related with k. When k is small, GPU-LSH cannot fetch enough candidates to select good NN results. From this view, GENIE should be better than GPU-LSH since it has stable approximation ratio given different k.

We use similar method to determine the parameters for GPU-LSH and GENIE on the OCR dataset which is also in Appendix I.2. GPU-LSH uses constant memory of the GPU to store random vector for LSH. Due to this implementation, the number of hash functions on OCR data cannot be larger than 8 otherwise the constant memory is overflowed. We use only 1M data points from the OCR dataset for GPU-LSH



(a) OCR (b) SIFT (c) DBLP (d) Tweets (e) Adult

Figure 8: The effectiveness of c-PQ.

Table 3: Accuracy of top-1 search on DBLP dataset for one round search process on the GPU (query length=40)

Percent of modified	0.05	0.1	0.15	0.2
Accuracy	1.0	0.9844	0.9707	0.9277

since it cannot work on larger dataset. we increase the number of hash tables (with fixing the number of hash functions as 8) until it can achieve similar prediction performance as GENIE as reported in Table 2 where the number of hash tables for GPU-LSH is set as 100. Note that the prediction performance of GPU-LSH is slightly worse than the one of GENIE. It is possible to improve the performance of GPU-LSH by increasing the number of hash tables, which will dramatically increase the running time for queries.

7.4.2 *Sequence similarity search of GENIE*

As we mentioned in Section 6.1, after finishing one round of top-k search on GENIE, we can identify that some of the queries do not obtain the real top-k results. Table 3 shows the percent of the queries obtaining correct top-1 similarity search results with one round of the search process for 1024 queries while we keep all the query length as 40. As we see from Table 3, with less than 10% modification, GENIE can return correct results for almost all the queries. Event with 20% modification, GENIE can still return correct results for more than 90% of queries. A typical application of such sequence similarity search is sequence error correction, where GENIE can return the most similar words (within minimum edit distance in a database) very quickly with acceptable accuracy. AppGram may do this job with better accuracy, but it has much larger latency.

8. CONCLUSION

In this paper, we presented GENIE, a generic inverted index framework for multiple similarity search queries on the GPU. GENIE can support the tolerance-Approximate Nearest Neighbour (τ-ANN) search for various data types under any similarity measure satisfying a generic Locality Sensitive Hashing scheme, as well as similarity search on original data with a "Shotgun and Assembly" scheme. In order to process more queries simultaneously in a batch on the GPU, we proposed a novel data structure – the Count Priority Queue, which reduces the memory requirement as well as the time cost significantly for multi-query processing. In particular, we investigate how to use GENIE to support ANN search in kernel space and in high dimensional space, similarity search on sequence data and short document data, and top-k selection on relational data. Extensive experiments on various datasets demonstrate the efficiency and effectiveness of GENIE. In future, we plan to extend our system to support more complex data types such as graphs and trees.

9. REFERENCES

[1] T. Alabi, J. D. Blanchard, B. Gordon, and R. Steinbach. Fast k-selection algorithms for graphics processing units. *Journal of Experimental Algorithmics (JEA)*, 17:4–2, 2012.

[2] A. Andoni. *Nearest neighbor search: the old, the new, and the impossible*. PhD thesis, Massachusetts Institute of Technology, 2009.

[3] A. Andoni and P. Indyk. E2lsh 0.1 user manual. `http://www.mit.edu/~andoni/LSH/manual.pdf`, 2005.

[4] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *PVLDB*, 4(8):470–481, 2011.

[5] S. Aparicio, J. Chapman, E. Stupka, N. Putnam, J.-m. Chia, P. Dehal, A. Christoffels, S. Rash, S. Hoon, A. Smit, et al. Whole-genome shotgun assembly and analysis of the genome of fugu rubripes. *Science*, 297(5585):1301–1310, 2002.

[6] A. Appleby. Smhasher & murmurhash. `http://code.google.com/p/smhasher/`, 2015.

[7] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103, 2010.

[8] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[9] M. Brown and D. G. Lowe. Automatic panoramic image stitching using invariant features. *International journal of computer vision*, 74(1):59–73, 2007.

[10] X. Cao, S. C. Li, and A. K. Tung. Indexing dna sequences using q-grams. In *Database Systems for Advanced Applications*, pages 4–16, 2005.

[11] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388, 2002.

[12] Y. Chen, A. Wan, and W. Liu. A fast parallel algorithm for finding the longest common sequence of multiple biosequences. *BMC bioinformatics*, 7(Suppl 4):S4, 2006.

[13] C. Cuda. Programming guide. *NVIDIA Corporation*, 2015.

[14] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*, pages 253–262, 2004.

[15] L. Devroye, P. Morin, and A. Viola. On worst-case robin hood hashing. *SIAM Journal on Computing*, 33(4):923–936, 2004.

[16] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance ir query processing. In *WWW*, pages 421–430, 2009.

[17] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIMOD*, pages 541–552, 2012.

[18] I. García, S. Lefebvre, S. Hornus, and A. Lasram. Coherent parallel hashing. In *ACM Transactions on Graphics (TOG)*, volume 30, page 161, 2011.

[19] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, D. Srivastava, et al. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[20] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[21] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics

processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21, 2009.

[22] X. He, D. Agarwal, and S. K. Prasad. Design and implementation of a parallel priority queue on many-core architectures. In *HiPC*, pages 1–10, 2012.

[23] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt. Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems. In *ISPASS*, pages 88–98, 2012.

[24] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4):11, 2008.

[25] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *FOCS*, pages 189–197, 2000.

[26] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

[27] T. Jaakkola, M. Diekhans, and D. Haussler. Using the fisher kernel method to detect remote protein homologies. In *ISMB*, volume 99, pages 149–158, 1999.

[28] H. Jegou, M. Douze, and C. Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *ECCV*, pages 304–317, 2008.

[29] J. Kärkkäinen. Computing the threshold for q-gram filters. In *Algorithm TheorySWAT*, pages 348–357. 2002.

[30] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, pages 339–350, 2010.

[31] J. Kim, S.-G. Kim, and B. Nam. Parallel multi-dimensional range query processing with r-trees on gpu. *Journal of Parallel and Distributed Computing*, 73(8):1195–1207, 2013.

[32] G. Kollias, M. Sathe, O. Schenk, and A. Grama. Fast parallel algorithms for graph similarity and matching. *Journal of Parallel and Distributed Computing*, 74(5):2400–2410, 2014.

[33] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *ICCV*, pages 2130–2137, 2009.

[34] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(6):1092–1104, 2012.

[35] A. Laarman, J. Van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In *FMCAD*, pages 247–256, 2010.

[36] M. Levandowsky and D. Winter. Distance between sets. *Nature*, 234(5323):34–35, 1971.

[37] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[38] M. Lichman. UCI machine learning repository. `http://archive.ics.uci.edu/ml`, 2013.

[39] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, 2004.

[40] N. Lukač and B. Žalik. Fast approximate k-nearest neighbours search using gpgpu. In *GPU Computing and Applications*, pages 221–234. 2015.

[41] L. Luo, M. D. Wong, and L. Leong. Parallel implementation of r-trees on the gpu. In *ASP-DAC*, pages 353–358, 2012.

[42] M. Moazeni and M. Sarrafzadeh. Lock-free hash table on graphics processors. In *SAAHPC*, pages 133–136, 2012.

[43] G. Navarro and R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2):1, 1998.

[44] J. Pan and D. Manocha. Fast gpu-based locality sensitive hashing for k-nearest neighbor computation. In *ACM SIGSPATIAL GIS*, pages 211–220, 2011.

[45] J. Pan and D. Manocha. Bi-level locality sensitive hashing for k-nearest neighbor computation. In *ICDE*, pages 378–389, 2012.

[46] M. Patil, S. V. Thankachan, R. Shah, W.-K. Hon, J. S. Vitter, and S. Chandrasekaran. Inverted indexes for phrases and strings. In *SIGIR*, pages 555–564, 2011.

[47] A. Rahimi and B. Recht. Random features for large-scale kernel machines. In *NIPS*, pages 1177–1184, 2007.

[48] D. Ravichandran, P. Pantel, and E. Hovy. Randomized algorithms and nlp: using locality sensitive hash function for high speed noun clustering. In *ACL*, pages 622–629, 2005.

[49] P. Russom et al. Big data analytics. *TDWI Best Practices Report, Fourth Quarter*, 2011.

[50] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *PVLDB*, 5(5):430–441, 2012.

[51] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, 1987.

[52] X. She, Z. Jiang, R. A. Clark, G. Liu, Z. Cheng, E. Tuzun, D. M. Church, G. Sutton, A. L. Halpern, and E. E. Eichler. Shotgun sequence assembly and recent segmental duplications within the human genome. *Nature*, 431(7011):927–930, 2004.

[53] S. Sonnenburg, V. Franc, E. Yom-Tov, and M. Sebag. Pascal large scale learning challenge. `http://largescale.ml.tu-berlin.de/instructions/`, 2008.

[54] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Combinatorial Pattern Matching*, pages 50–63, 1996.

[55] S. Tatikonda and S. Parthasarathy. Hashing tree-structured data: Methods and applications. In *ICDE*, pages 429–440, 2010.

[56] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in distributed text document retrieval systems. In *PDIS*, pages 8–17, 1993.

[57] T. T. Tran, M. Giraud, and J.-S. Varre. Perfect hashing structures for parallel similarity searches. In *IPDPSW*, pages 332–341, 2015.

[58] G. Wang, B. Wang, X. Yang, and G. Yu. Efficiently indexing large sparse graphs for similarity search. *TKDE*, 24(3):440–451, 2012.

[59] J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. *arXiv preprint arXiv:1408.2927*, 2014.

[60] X. Wang, X. Ding, A. K. Tung, S. Ying, and H. Jin. An efficient graph indexing method. In *ICDE*, pages 210–221, 2012.

[61] X. Wang, X. Ding, A. K. Tung, and Z. Zhang. Efficient and effective knn sequence search with approximate n-grams. *PVLDB*, 7(1):1–12, 2013.

[62] D. Wu, F. Zhang, N. Ao, G. Wang, J. Liu, and J. Liu. Efficient lists intersection by cpu-gpu cooperative computing. In *IPDPSW*, pages 1–8, 2010.

[63] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD*, pages 766–777, 2005.

[64] R. Yang, P. Kalnis, and A. K. Tung. Similarity evaluation on tree-structured data. In *SIGMOD*, pages 754–765, 2005.

[65] X. Ying, S.-Q. Xin, and Y. He. Parallel chen-han (pch) algorithm for discrete geodesics. *ACM Transactions on Graphics*, 33(1):9, 2014.

[66] S. You, J. Zhang, and L. Gruenwald. Parallel spatial query processing on gpus using r-trees. In *ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data*, pages 23–31, 2013.

[67] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge & Data Engineering*, 27(7):1920–1948, 2015.

[68] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *PVLDB*, 8(11):1226–1237, 2015.

[69] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *SIGMOD*, pages 145–156. ACM, 2002.

[70] J. Zhou and A. K. Tung. Smiler: A semi-lazy time series prediction system for sensors. In *SIGMOD*, pages 1871–1886, 2015.

[71] Y. Zhou and J. Zeng. Massively parallel a* search on a gpu. In *AAAI*, pages 1248–1255, 2015.

APPENDIX

A. GRAPHICS PROCESSING UNIT

The Graphics Processing Unit (GPU) is a device that shares many aspects of Single-Instruction-Multiple-Data (SIMD) architecture. The GPU provides a massively parallel execution environment for many threads, with all of the threads running on multiple processing cores, and executing the same program on separate data. We implemented our system on an NVIDIA GPU using the Compute Unified Device Architecture (CUDA) toolkit [13]. Each CUDA function is executed by an array of *threads*. A small batch (e.g. 1024) of threads is organized as a *block* that controls the cooperation among threads.

B. LOAD BALANCE

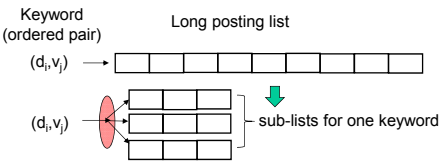


Figure 10: Splitting long posting list for load balance.

For the inverted index, there may be some extreme long posting lists which can become the bottleneck of the system. Thus, it is necessary to consider the load balance problem in such application context. In our system, we also implement a load balance function, whose solution is to limit the length of posting lists and the maximum elements to be processed by one block. When the posting list is too long, we divide such a long posting list into a set of sublists, then instead of using a bijective map, we build a one-to-many map to store the addresses of the sub-lists in the List Array. Figure 10 gives an illustration for splitting a long posting list to three posting sub-lists. During scanning the (sub-)posting lists, we also limit the number of lists processed by one block. In our system, after enabling the load balance function, we limit the length of each (sub-)posting list as 4K and each block takes two (sub-)posting lists at most. It is worthwhile to note that, if there are already many queries running on the system, the usefulness of load balance is marginally decreased. This is because all the computing resources of the GPU have been utilized when there are many queries and the effect of load balance becomes neglected.

B.1 Experimental study on load balance

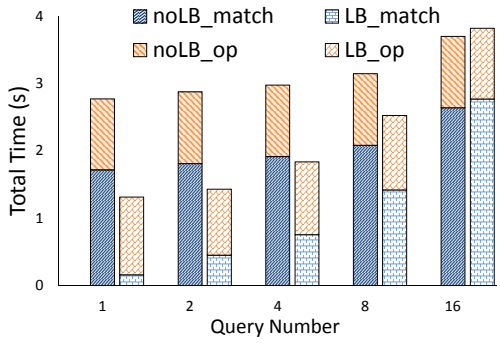


Figure 11: Load balance on Adult data (with 100M data points)

We study the effect of the load balance for GENIE in this section. We use the Adult dataset which has long posting lists since some its attributes have only a few of categories (e.g. sex). We also duplicate the Adult dataset to 100M points to show the effect more clearly. In this experiment, we exert exact match for all attributes given a query, and return the best match candidates to the query. For load balance, we limit the length of each sub-list as 4096, and let each block of the GPU take two (sub-)lists at most. Figure 11 illustrates the running time of GENIE with and without load balance by varying the number of queries. The "noLB_match" and "LB_match" represent the running time with and without enabling load balance function respectively. The "noLB_op" and "LB_op" represent other operation cost for searching including transferring queries and results. From Figure 11, we can see that the load balance function can effectively allocate the workload to different blocks by breaking down the long lists. With increasing of the number of queries, the effect of the load balance is marginally decreased. The reason is that when the number of queries is larger, GENIE has already maximized the possibility for parallel processing by using one block for one posting list. For 14-dimensional dataset with 16 queries, all the stream processors of the GPU have been utilized. Besides, since the load balance requires some additional cost to maintain the index, the running time of GENIE with load balance is slightly higher than the one without load balance when the GPU is fully utilized.

C. K-SELECTION ON THE GPU AS PRIORITY QUEUE

We use a k-selection algorithm from a array to construct a priority queue-like data structure on the GPU, which is named as SPQ in the paper. To extract the top-k object from an array, we modify a GPU-based bucket-selection algorithm [1] for this purpose. Figure 12 shows an example for such selection process.

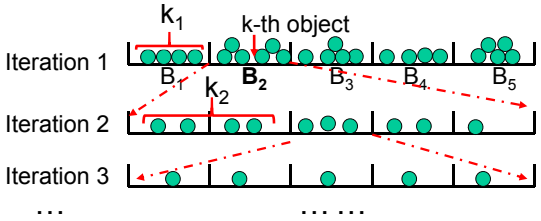


Figure 12: Example for bucket k-selection. The iteration repeats until $k = k_1 + k_2 + ... + k_t$.

The algorithm has multiple iterations, and each iteration has three main steps. Step (1): we use a partition formulae $bucket_{id} = \lfloor (count - min)/(max - min) * bucket\_num) \rfloor$ to assign every objects into buckets. In Figure 12, all the objects in a hash table are assigned bucket B_1 to B_5. Step (2): We then check the bucket containing $k - th$ object. In Figure 12, B_2 is the selected bucket in Iteration 1. Step (3): we save all the objects before the selected bucket, and denote the number of saved objects as k_i (e.g. k_1 in Iteration 1). Then we repeat Step (1)-(3) on the objects of the selected buckets until we find all the top-k objects (i.e. $k = k_1 + k_2 + ... + k_t$). From the algorithm, we can see that this method requires to explicitly store the object ids to assign them into buckets. In regard to multiple queries, we use one block to handle one hash table to support parallel selection

in the GPU-based implementation. In our experiment, the algorithm usually finishes in two or three iterations.

D. ROBIN HOOD HASHING

The general idea of the Robin Hood hashing is to record the number of probes for reaching (inserting or accessing) an entry due to conflicts. We refer to this probing number as age. During the insertion, one entry will evict an existing entry in the probed location if the existing entry has a smaller age. Then we repeat the process of inserting the evicted entry into the hash table until all the entries are inserted. For a non-full table of size T, the expected maximum age per insertion or access is $\Theta(log_2 logT)$ [15].

Since for each *AuditThreshold*, k objects can pass the Gate into the Hash Table, there are at most $O(k{*}max\_count\_value)$ objects in the Hash Table. Thus, the size of the Hash Table can be set as $O(k * max\_count\_value)$ multiplied by a load factor α which can be simply set as 1.5 indicating that 1/3 locations of the Hash Table are empty.

Another improvement of the Robin Hood Hashing is to take the race condition problem into account (which is not covered in [18] since the paper is for static data applications). The update and insertion of the gate and the hash table should take place in a critical section to avoid race conditions. The use of lock for critical section must be avoided since multiple threads in the same warp competing for locks will cause deadlock due to the SIMD architecture. We adopt a lock-free synchronization mechanism which is studied in [35, 42]. The idea is to use the atomic Compare and Swap (CAS) operation to guarantee that only one thread can update the critical section but all threads within one warp can complete the operation in a finite number of steps.

E. VIEW THE GENIE FROM MLE

The relationship between the collision of LSH and the similarity measure can be explained from the view of the maximum likelihood estimation (MLE). Given a point p and a query q with a set of LSH functions $\mathbb{H} = \{h_1, h_2, ..., h_m\}$ defined by Eqn.(1), if there are c functions in $\mathbb{H}$ satisfying $h_i(p) = h_i(q)$, we can obtain an unbiased estimation of $sim(p, q)$ by maximum likelihood estimation (MLE):

$$\hat{s} = \frac{c}{m} \tag{11}$$

Eqn. 11 provides an appealing insight that we can estimate the real $sim(p, q)$ according to the collision number of hash functions in the hash set $\mathbb{H}$ (Eqn. 11 has been discussed in [50]). In fact, Eqn. 11 has been implicitly used to tune the number of LSH functions [11, 48]. This is good news since our GENIE system based on the count-match model can effectively support counting the number of collision functions. However, we should also analyze the error bound between the estimate $\hat{s}$ and the real similarity measure $s = sim(p, q)$.

F. PROOF

F.1 Proof of Theorem 4.1

Before giving the proof of the theorem, we first illustrate the following lemma.

LEMMA F.1. *In Algorithm 1, after finishing all the updates of the Gate, we have $ZipperArray[AuditThreshold] < k$ and $ZipperArray[AuditThreshold - 1] \geq k$.*

PROOF. In Algorithm 1, after each update of *ZipperArray* in line 5, we check whether $ZipperArray[AuditThreshold] \geq k$ in line 6. If it is, we increase *AuditThreshold* (in line 7). Therefore, we always have $ZipperArray[AuditThreshold] < k$. Similarly, since we only increase *AuditThreshold* in line 7, we can guarantee that $ZipperArray[AuditThreshold - 1] \geq k$. □

Then we can prove the Theorem 4.1.

PROOF. For the claim that "the top-k candidates are only stored in the Hash Table (not in Bitmap Counter)", since there at least $O(k)$ objects passing the gate for each possible value of AuditThreshold, therefore, the objects left in Bitmap Counter cannot be the top-k candidates. For the same reason, the objects in the Hash Table is $O(k * AuditThreshold)$.

For the claim that "$MC_k = AuditThrehsold - 1$", we prove it by contradiction. On the one hand, if we suppose $MC_k > AuditThrehsold - 1$, we can deduce that $MC_k \geq AuditThrehsold$. Therefore, we can also further infer that $ZipperArray[AuditThreshold] \geq k$, which contradicts with Lemma F.1. On the other hand, if we suppose $MC_k < AuditThrehsold - 1$, then there must be less than k objects with match count greater than or equal to $AuditThreshold - 1$, which contradicts with Lemma F.1. Therefore, we only have $MC_k = AuditThrehsold - 1$. □

F.2 Proof of Theorem 5.1

PROOF. The proof of Theorem 5.1 is inspired by the routine of the proof of Lemma 4.2.2 in [2]. For convenience, let c be the the return of match-count model $c = MC(Q_q, O_p)$, which essentially is the number of hash function $f_i(\cdot)$ such that $f_i(p) = f_i(q)$. The collisions of $f_i(\cdot)$ can be divided into two classes: one is caused by the collision of the LSH $h_i(\cdot)$ (since if $h_i(p) = h_i(q)$ then we must have $f_i(p) = f_i(q)$), the other one is caused by the collision of the random projection (meaning $h_i(p) \neq h_i(q)$ but $r_i(h_i(p)) = r_i(h_i(q))$). Therefore, we further decompose count c as $c = c_h + c_r$ where c_h denotes the number of collisions of caused by $h_i(\cdot)$ and c_r denotes the one caused by $r_i(\cdot)$ when $h_i(p) \neq h_i(q)$.

We first prove that that $|c_h/m - sim(p, q)| \leq \epsilon/2$ with probability at least $1-\delta/2$ given $Pr[h(p) = h(q)] = sim(p, q)$. This is deduced directly by Hoeffding's inequity if $m = 2\frac{ln(3/\delta)}{\epsilon^2}$, which is:

$$Pr[|c_h/m - sim(p, q)| \geq \epsilon/2] \leq 2e^{-2m(\frac{\epsilon}{2})^2} = \frac{2\delta}{3} \tag{12}$$

For the rest of the $m - c_h$ hashing functions, we need to prove that the collision $c_r \leq (\omega + \epsilon/2)m$ with probability at least $1 - \delta/3$, where ω is the collision probability of r_i. To simplify the following expression, we also denote $\epsilon/2$ by β, i.e. $\beta = \epsilon/2$. Note that the expectation of c_r is $E(c_r) = \omega(m - c_h)$. According to the Hoeffding's inequality, we have $Pr[c_r > (\omega + \beta)m] = Pr[c_r > (\omega + \frac{\omega c_h + \beta m}{m - c_h})(m - c_h)] \leq e^{-2(\frac{\omega c_h + \beta m}{m - c_h})^2 (m - c_h)}$. We have $(\frac{\omega c_h + \beta m}{m - c_h})^2 (m - c_h) \geq (\frac{\beta m}{m - c_h})^2 (m - c_h) \geq \frac{\beta^2 m^2}{m - c_h} \geq \beta^2 m$. Therefore, we have $e^{-2(\frac{\omega c_h + \beta m}{m - c_h})^2 (m - c_h)} \leq e^{-2\beta^2 m}$. Finally, we have $Pr[c_r/m > (\omega + \beta)] \leq e^{-2\beta^2 m}$. Since $\beta = \epsilon/2$ and $m = 2\frac{ln(3/\delta)}{\epsilon^2}$, we have $Pr[c_r/m > (\omega + \epsilon/2)] \leq \frac{\delta}{3}$. To combine above together, we have:

$$
\begin{aligned}
&Pr[|c_h/m + c_r/m - sim(p,q)| > \omega + \epsilon] \\
&\leq Pr[(|c_h/m - sim(p,q)| > \epsilon/2) \cup (c_r/m > \omega + \epsilon/2)] \\
&\leq Pr[(|c_h/m - sim(p,q)| > \epsilon/2)] + Pr[(c_r/m > \omega + \epsilon/2)] \\
&\leq \delta
\end{aligned}
$$

We also have $\omega = D*1/D^2 = 1/D$, therefore, $|MC(Q_q, O_p) - sim(x,y)| = |(c_h + c_r)/m - sim(p,q)| \leq \epsilon + 1/D$ with probability at least $1-\delta$. □

F.3 Proof of Theorem 5.2

PROOF. For convenience, we denote that the output count values of match-count model as $c = MC(Q_q, O_p)$ and $c^* = MC(Q_q, O_{p^*})$, and denote the real similarity measures as $s = sim(p,q)$ and $s^* = sim(p^*, q)$. We can get that

$$
\begin{aligned}
&Pr[|c/m - s| \leq \epsilon \cap |c^*/m - s^*| \leq \epsilon] \\
&= Pr[|c/m - s| \leq \epsilon] \cdot Pr[|c^*/m - s^*| \leq \epsilon] \\
&\geq (1-\delta)(1-\delta) \geq 1 - 2\delta
\end{aligned}
$$

We also have that $c \geq c^*$ (c is top result) and $s^* \geq s$ (s^* is true NN). From $|c/m - s| \leq \epsilon$ and $|c^*/m - s^*| \leq \epsilon$, we can get that $s^* \leq c^*/m + \epsilon$ and $s \geq c/m - \epsilon$, which implies that $0 \leq s^* - s \leq c^*/m + \epsilon - (c/m - \epsilon) \leq 2\epsilon$.

To sum up, we can obtain that $Pr[|sim(p^*,q) - sim(p,q)| \leq 2\epsilon] \geq 1 - 2\delta$. □

F.4 Proof of Theorem 6.1

PROOF. The observation between the count of shared n-gram and edit distance has been studied in [54, 19]. Note that if the same n-gram occurs c_s times in S and c_q times in Q, we count the common n-gram number as $min(c_s, c_q)$ [29]. □

F.5 Proof of Lemma 6.2

PROOF. We prove it by contradiction. Suppose the real k-th similar sequence S^{k*} cannot be found by GENIE. The edit distance between the query Q and the real k-th similar sequence S^{k*} is τ_{k^*}. Then the match count between S^{k*} and Q satisfies: $c_{k^*} \geq max\{|Q|, |S^{k*}|\} - n + 1 - \tau_{k^*} * n$. We also have $\tau_{k^*} \leq \tau_{\mathbf{k}'}$, then we can get $c_{k^*} \geq |Q| - n + 1 - \tau_{k^*} * n \geq |Q| - n + 1 - \tau_{\mathbf{k}'} * n > c_{\mathbf{K}}$. However, if $c_{k^*} > c_{\mathbf{K}}$, sequence S^{k*} must have already been retrieved by GENIE. Therefore, the real k-th similar sequence S^{k*} can be found by GENIE. In the same way, we can prove that all the top-k results can be correctly returned by GENIE under the premise condition. □

G. RANDOM BINNING HASHING

For any kernel function $k(\cdot)$ satisfying that $p(\sigma) = \sigma k''(\sigma)$ (where $k''(\cdot)$ is the second derivative of $k(\cdot)$) is a probability distribution function on $\sigma \geq 0$, we can construct an LSH family by random binning hashing (RBH) for kernel function $k(\cdot)$. For each RBH function $h(\cdot)$, we impose a randomly shift regular grid on the space with a grid cell size g that is sampled from $p(\sigma)$, and a shift vector $u = [u^1, u^2, ..., u^d]$ that is drawn uniformly from $[0, g]$. For a d-dimensional point $p = [p^1, p^2, ..., p^d]$, the hash function is defined as:

$$
h(p) = [\lfloor (p^1 - u^1)/g \rfloor, ..., \lfloor (p^d - u^d)/g \rfloor] \tag{13}
$$

The expected collision probability of RBH function is $Pr(h(p) = h(q)] = k(p,q)$. One typical kernel function satisfying the conditions for RBH is Laplacian kernel $k(p,q) = exp(- \parallel p - q \parallel_1 /\sigma)$. We refer interested readers to [47]. For the rehashing mechanism on GENIE, we select MurmurHashing3 [6] as the random projection function.

H. LSH IN HIGH DIMENSIONAL SPACE

In this section, we discuss ANN search in high dimensional space, where, instead of using Eqn. 1, an LSH function family is usually defined as follows [14]:

DEFINITION H.1. *In a d-dimensional $l_{\mathfrak{p}}$ norm space R^d, a function family $\mathbb{H} = \{h : R^d \to [0, D)\}$ is called $(r_1, r_2, \rho_1, \rho_2)$-sensitive if for any $p, q \in R^d$:*

- *if $\parallel p - q \parallel_{\mathfrak{p}} \leq r_1$, then $Pr[h(p) = h(q)] \geq \rho_1$*
- *if $\parallel p - q \parallel_{\mathfrak{p}} \geq r_2$, then $Pr[h(p) = h(q)] \leq \rho_2$*

where $r_1 < r_2$ and $\rho_1 > \rho_2$ and $\parallel p - q \parallel_{\mathfrak{p}}$ is distance function in $l_{\mathfrak{p}}$ norm space.

Based on the $\mathfrak{p}$-stable distribution [25], an LSH function family in $l_{\mathfrak{p}}$ norm space can be defined as [14]:

$$
h(q) = \lfloor \frac{\mathbf{a}^T \cdot q + b}{w} \rfloor \tag{14}
$$

where $\mathbf{a}$ is a d-dimensional random vector whose entry is drawn independently from a $\mathfrak{p}$-stable distribution for $l_{\mathfrak{p}}$ distance function (e.g. Gaussian distribution for l_2 distance), and b is a random real number drawn uniformly from $[0, w)$.

A similar counting method has also been used for c-ANN search in [17] where a Collision Counting LSH (C2LSH) scheme is proposed for c-ANN search. Though our method has different theoretical perspective from C2LSH, the basic idea behind them is similar: the more collision functions between points, the more likely that they would be near each other. From this view, the C2LSH can corroborate the effectiveness of the τ-ANN search of our method.

I. MORE EXPERIMENT

I.1 Average running time per query

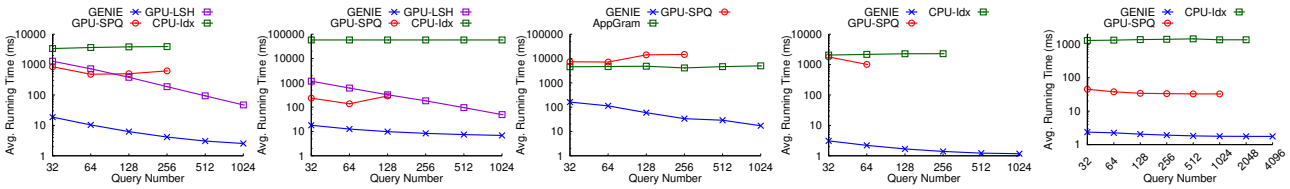
Figure 13 (y-axis is log-scaled) shows the average running time per query with varying numbers of queries, which reveals more insight about the advantage of GENIE. On the one hand, GENIE can run more queries for one batch which gives us the opportunities to amortize the time cost for multi-query similarity search. On the other hand, both the slope and the value of the curve of GPU-LSH are larger than the ones of GENIE, which means GENIE is good at fully utilizing the parallel computational capability of the GPU given multiple queries.

I.2 Metric and parameter setting for ANN search on the GPU

We first give a definition of "approximation ratio". Formally, for a query point q, let $\{p_1, p_2, ..., p_k\}$ be the ANN search results sorted in an ascending order of their $l_{\mathfrak{p}}$ normal distances to q. Let $\{p_1^*, p_2^*, ..., p_k^*\}$ be the true kNNs sorted in an ascending order of their distances to q. Then the approximation ratio is formally defined as:

$$
\frac{1}{k} \sum_{i=1}^{k} \frac{\|p_i - q\|_{\mathfrak{p}}}{\|p_i^* - q\|_{\mathfrak{p}}} \tag{15}
$$

In the experiment evaluation (especially for running time) for ANN search in high dimensional space on the SIFT data



(a) OCR (b) SIFT (c) DBLP (d) Tweets (e) Adult

Figure 13: Average running time per query.

set, we set the parameters of GPU-LSH and GENIE to ensure that they have similar approximation ratio. For ANN search of GENIE, we set the number of hash functions as 237 which is determined by setting $\epsilon = \delta = 0.06$ (as discussed in Section 5.1.2 and illustrated by Figure 5). Another parameter for ANN search in high dimensional space is the bucket width (of Eqn. 8). According to the method discussed in the original paper of E2LSH [14], we divide the whole hash domain into 67 buckets. The setting of bucket width is a trade-off between time and accuracy: relative larger bucket width can improve the approximation ratio of GENIE, but requires longer running time for similarity search.

For GPU-LSH, there are two important parameters: the number of hash functions per hash table and the number of hash tables. With fixing the number of hash tables, for different settings of the number of hash functions, we find GPU-LSH has the minimal running time to achieve the same approximation ratio when the number of hash functions is 32. After fixing the number of hash functions as 32, we gradually increase the number of hash tables for GPU-LSH, until it can achieve approximation ratio similar to that of ANN search by GENIE (k is fixed as 100). The number of hash tables is set as 700.

For ANN search in Laplacian kernel space by GENIE, except parameters ϵ and δ which determine the number of hash functions, another parameter is the kernel width σ of the Laplacian kernel $k(x, y) = exp(-\|x - y\|_1/\sigma)$. We random sample 10K points from the dataset and use their mean of paired l_1 distance as the kernel width. This is a common method to determine the kernel width for kernel function introduced by Jaakkola et al.[27].