Generic Inverted Index on the GPU

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

ABSTRACT

Data variety, as one of the three Vs of the Big Data, is manifested by a growing number of complex data types such as documents, sequences, trees, graphs and high dimensional vectors. To perform similarity search on these data, 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 on the GPU (called GENIE), which can support similarity search of multiple queries on various data types. GENIE can effectively support the approximate nearest neighbor search in different similarity measures through exerting Locality Sensitive Hashing schemes, as well as similarity search on original data such as short document data and relational data. Extensive experiments on different real-life datasets demonstrate the efficiency and effectiveness of our system.

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 [50]. 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 [47, 37, 58, 63, 64, 11]. 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, 14, 57].

In this paper, we observe that similarity search on many of these data types can be supported by an inverted index and proceed to develop a generic inverted index on the GPU to speed up searching on a large variety of data types. The insight is that many data types can be transformed into a form that can be searched by an inverted-index-like structure. Such a transformation can be done by the Locality Sensitive Hashing (LSH) for several similarity measures [13, 16] and by Shotgun and Assembly [5, 53] for complex structured data like documents, sequences, trees and graphs. To support our claim, let us first try to show a common framework that is applicable for similarity search on a variety of complex data types.

- **High dimensional data.** High dimensional data points can be transformed by an LSH scheme [16] derived from the $\mathfrak{p}$-stable distribution in $l_\mathfrak{p}$ space. In such a scheme, multiple hash functions are used to hash the data points into different buckets and points that are frequently hashed to the same bucket are deemed to be similar. As such, we can build an inverted index where each posting list corresponds to a list of points that are hashed to a particular bucket. Given a query point, Approximate Nearest Neighbour (ANN) search can be performed by hashing the query point based on the hash functions and then scanning the corresponding posting list to find data points that occur in many of these buckets.
- **Sets, feature sketches and geometries.** 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 [13]. Similarity search on such data can also be transformed by Locality Sensitive Hashing functions [13] and searched through an inverted index.
- **Documents, sequences, trees and graphs.** These complex structure data can be transformed within the "Shotgun and Assembly" [5, 53] framework. Specifically, the data will be broken down into smaller sub-units ("shotgun"), such as words for documents [56, 44], q-grams for sequences [38, 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].
- **Relational data.** The inverted index for relational

.

data can be built with the similar “Shotgun and Assembly” framework of complex structured data. 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 the attribute-value 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.

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[1] 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 inverted index, 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. At the time of writing, 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[2]. 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 [66]. 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 match, scan and count.

In view of the above considerations, in this paper, we propose a Generic Inverted Index, called GENIE, which is implemented on the GPU to support similarity search in parallel. To cater to 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. The match-count model essentially performs scan on posting lists that are relevant to the similarity queries and maintains a frequency count for the number of times that a data object is seen in these posting lists. GENIE is designed to support efficient and parallel computation of the match-count model between multiple queries and a set of objects on the GPU.

To exhibit that our match-count model is a general model that can be used in similarity search for various data types with different measures (i.e. *generalization*), we first define the concept of Tolerance-Approximate Nearest Neighbour (τ-ANN) search for locality sensitive hashing 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 Locality Sensitive Hashing (LSH) scheme satisfying the definition of LSH given by Charikar [13]. We further demonstrate that GENIE can also perform ANN search in high dimensional space through exerting another popular LSH scheme given by Indyk and Motwani [26] based on the $\mathfrak{p}$-stable distribution. Through these derivations, we ensure that GENIE is a generic index that can support various data types on a generic LSH scheme. For complex data types without known LSH transformation, there is a choice of adopting the “Shotgun and Assembly” framework. We will also showcase this in this paper by performing similarity search on short document data and relational data using GENIE.

However, a straightforward implementation of the match-count model requires huge memory space, which limits the maximum number of parallel queries within a batch process on the GPU. Therefore, we propose a novel data structure on the GPU, called Count Heap, which can not only reduce the time cost for similarity search per query, but also reduce the memory requirement for multiple queries. By reducing the memory requirement, the deployment of the Count Heap can substantially increase the number of queries within a query processing batch on the GPU.

We summarize our contributions as follows:

- We propose a generic inverted index (GENIE) on the GPU, which can support similarity search of multiple queries under different measures (on different data types) on the GPU.
- With regard to the system design of the inverted index, we also devise a novel data structure, named Count Heap, which significantly reduces the memory requirement and 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 our system can effectively support the τ-ANN search with the help of the generic Locality Sensitive Hashing scheme. We also showcase the similarity search of multiple queries on short document data and relational data in original space by GENIE.
- 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. At 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. Finally, we will conduct experiment evaluation in Section 6 and will conclude the paper in Section 7.

2. RELATED WORK

2.1 Similarity search on complex Data

[1] 128 dimensions to be exact

[2] Looking forward, the Nvidia Pascal GPU is expected to be released next year with 16GB of HBM2 memory which has an access bandwidth of 1TB/s, three times the current bandwidth. [10]

2.1.1 High dimensional data

Due to the "curse of dimensionality", spatial index methods, like R tree [21], R+ tree [52] and R* tree [8], provide 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 Approximate Nearest Neighbor (ANN) search in high dimensional space. The theoretical foundation of ANN search is based on the Locality Sensitive Hashing (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 [16]. We refer interested readers to a survey of LSH [59].

2.1.2 Sets, feature sketches and geometries

Various types of data objects are generated due to the growth in information technologies. Example of such objects include page sets from web service, feature sketches from multimedia and geometries from computer graphics. 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 approximate nearest neighbor search in such kernel spaces has also drawn considerable attention. Similar as searching high dimensional points, the most notably solution for this problem is the Locality Sensitive Hashing (LSH). For example, Charikar [13] 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. Our GENIE system can support the similarity search in an arbitrary kernel space if it has a LSH scheme.

2.1.3 Documents, sequences, trees and graphs

There is a wealth of literature concerning the similarity search of complex structured data, and a large number of indexes have been devised. Many of them adopt the "Shotgun and Assembly" approach [5, 53] 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, 44], q-grams for sequences [38, 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. 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 approximate nearest neighbour (ANN) search by LSH. Pan and Manocha [45] propose a parallel LSH searching method on the GPU using a Bi-level LSH algorithm [46]. Lukac and Zalik[41] study the LSH method with multi-probe variant for ANN search using GPUs. Both of these two methods are specially designed for ANN search in the $l_{\mathfrak{p}}$ space. In contrast, our GENIE system can generally support LSH for ANN search with various similarity measures.

2.3 Data structures and indexes on the GPU

There are several works about specific indexes and data structures on the GPU, including the inverted index and tree-based structures. The existing papers [18, 62, 4] about an inverted index on the GPU may seem related to our work. Nevertheless, they focus on designing specialized algorithms for two important operations of search engines – list intersection and index compression, which cannot be generalized for similarity search. A two-level inverted-like index on the GPU is also studied for continuous time series search under the Dynamic Time Warping distance [69].

Tree-based data structures are also investigated to fully utilize the parallel capability on the GPU. Parallel accessing to the B-tree[68, 22, 7] or R-tree [42, 31, 65] 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 [67, 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 general structure of our inverted index.

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* $O = \{o_{i,1}, ..., o_{i,r}\}$*, we map each query item* q_i *to a natural integer* $C : (q_i, O) \rightarrow \mathbb{N}_0$ *where* $C(q_i, O)$ *returns the number of elements* $o_{i,j} \in O$ *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* $MC(Q, O) = \sum_{q_i \in Q} C(q_i, O)$*.*

Accordingly, 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 Inverted index for multiple queries

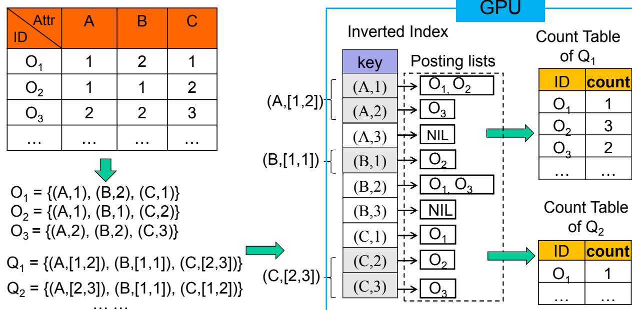


Figure 1: An illustration of the inverted index on a relational table. Multiple queries are processed, accessing the inverted index in parallel.

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 first give a brief introduction of the computational architecture of the GPU, and then overview our index method for facilitating the computation of the match-count model on the GPU.

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 [15]. 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.

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 Generic Inverted Index (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 with different similarity measures in Section 5.

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 Heap, which is a heap-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 on 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* on 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, but 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.

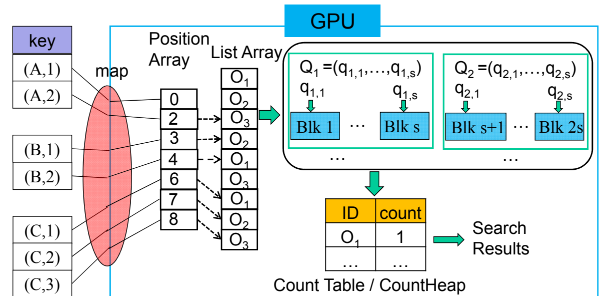


Figure 2: Overview of the inverted index and data flow on the GPU.

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 in scanning posting lists, which is introduced in next section.

4.1.1 Load balance

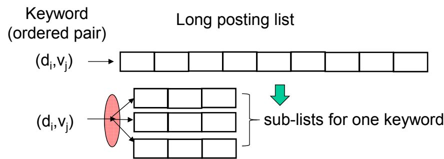


Figure 3: Splitting long posting list for load balance.

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 3 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.

4.2 The Count Heap

One drawback of the Count Table is its large space cost. We need to allocate 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. Such an high space requirement limits the number of queries being processed in a batch. In order to support more queries in a batch, we propose a novel data structure, called Count Heap, to replace the Count Table.

4.2.1 The structure of Count Heap

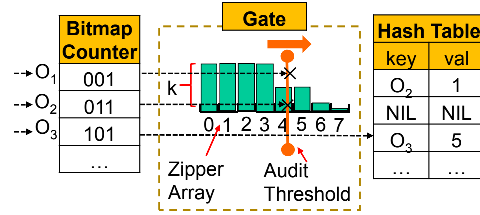


Figure 4: An illustration of the Count Heap.

Figure 4 shows three main components of the count-heap: (1) the Bitmap Counter to store the count for each object, (2) the Gate to filter candidates, and (3) the Hash Table to store the count for the final candidates.

To save space, we first compress the whole Count Table in a bitmap structure. One unique property of the Count Table is that the maximum count of each object is bounded. Take range queries in relational tables as an example. The maximum count is no more than the number of attributes of the table. To leverage this property, we use a bitmap to store the count instead of an integer. The size of the bitmap counter is equal to the number of objects multiplied by the number of bits necessary to encode the maximum value of the count. In this way, the identifier of an object is implicitly expressed by the address of the bitmaps, and the value of the counter is stored by a binary code.

In order to select the top-k objects with the highest count, one possible solution is to employ a k-selection algorithm, such as [1], on the GPU. However, such the k-selection method requires to store object id explicitly. Taking the example of 10M points again, if we want to submit a batch of one thousand queries, except the size of bitmap structures, the required space to store the object id of the method in [1] is already 40 GB (by allocating one integer for object id, the size is $1k(queries) \times 10M(points) \times 4(bytes) = 40GB$). We cannot save space by bitmap method if we adopt such k-selection method. We refer readers to Appendix A for a brief explanation of the k-selection method on the GPU.

To reduce the space requirement of selecting the top-k objects with the highest count, we further introduce the other two components in the Count Heap: the Gate and the Hash Table. The general idea is that, during the updating process of the bitmap counter, we also insert the a small number of candidate objects into the Hash Table if the object's count is larger than an adaptive threshold (maintained in the Gate). The objective is that, after scanning the posting lists, only the objects in the Hash Table can be the potential candidates of the top-k results, while all the objects in the Bitmap Counter can be safely abandoned.

The Gate has a *ZipperArray* whose size is equal to the maximum value of the count. *ZipperArray[i]* records the number of objects whose counts have reached i. As we need to select the top-k objects, we maintain a threshold called *AuditThreshold* to restrict objects from going to the Hash Table.

4.2.2 Updating the Count Heap

We present the update process of the Count Heap in Algorithm 1. Figure 4 also gives an illustration of the updating process. When the count of an object is updated in the Bitmap Counter, at the same time, we further 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 4 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 use the new entry to replace the old 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 Heap

```
   // 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
```

4.2.3 Selecting top-k objects from the Count Heap

After updating the Gate (which happens at the same time with scanning the inverted index), we claim that all the real top-k candidates are stored in the Hash Table, since at least k objects passed the gate for each possible value of *AuditThreshold*. It seems that we still need a dedicated method to extract the top-k objects from the Hash Table. However, in Theorem 4.1, we prove that the match count value of the $k-$th object is just $AuditThreshold - 1$, which reduces the candidate size significantly. Before giving the proof of the theorem, we first illustrate the following lemma.

LEMMA 4.1. *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 $ZipperArray[AuditThreshold - 1] \geq k$. □

THEOREM 4.1. *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. We prove it by contradiction. On the one hand, if we suppose $MC_k > AuditThrehsold - 1$, we can deduce that $MC_k \geq AuditThrehsold$. Thus, we can further infer that $ZipperArray[AuditThreshold] \geq k$, which contradicts with Lemma 4.1. 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 4.1. Therefore, we only have $MC_k = AuditThrehsold - 1$. □

With the help of Theorem 4.1, to select the top-k objects, we only need to scan the Hash Table to select all objects with match count greatler than ($AuditThrehsold - 1$). If there are multiple objects with match count equal to ($AuditThrehsold - 1$), we break ties randomly.

4.2.4 Hash Table with modified Robin Hood Scheme

The hashing operations will be carried out on GPU devices, which are SIMD machines where the total execution time of a set of instructions depends on the slowest thread. The implication of this consideration is that the entry with the longest conflict chain delays everyone. So we need to focus on solving the conflict issue in hashing. As a result, we adopt the Robin Hood open address hashing scheme [12, 17] for the implementation of the Hash Table. A similar GPU-based hash table implementation with Robin Hood scheme has been studied in [20].

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 $\Theta(log_2 log T)$ [17].

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.

We further improve the Robin Hood hashing scheme. The general idea is that all the entries with values smaller than $(AuditThrehsold - 1)$ in the Hash Table cannot be the top-k candidates (see theorem 4.1), therefore, they can be safely replaced by the inserted entry regardless of ages. If the value of the existing entry is smaller than the $(AuditThrehsold - 1)$, we can directly overwrite the entry by the inserted one. In this way, we can significantly reduce the probe times of the insertion and access operations of the Hash Table, since many inserted keys become expired with the monotonous increase of *AuditThreshold*.

Another improvement of the Robin Hood Hashing is to take the race condition problem into account (which is not covered in [20] 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, 43]. 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.

5. GENERIC SEARCH ON THE INDEX

In this section, we discuss how to generalize our match-count model for similarity search of various data types with different measures. We first show that GENIE (with match-count model) can support the ANN search for any similarity measure which has a generic Locality Sensitive Hashing (LSH) scheme. For complex data types which have no LSH transformation, we also showcase the "Shotgun and Assembly" framework of GENIE by performing similarity search on short document data and relational data.

5.1 ANN search by generic similarity measure

The general Nearest Neighbor Search (NNS) problem is:

Given a set of n points $P = \{p_1, p_2, .., p_n\}$ [3] in a space S, retrieve a point in P that is closest to a query point $q \in S$ under a similarity measure $sim(p_i, q)$. It is often computationally infeasible and unnecessary to retrieve the exact nearest neighbor. Therefore, the 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. One of the most popular solutions for the ANN search problem is Locality Sensitive Hashing [26, 13, 59].

In order to integrate existing locality sensitive hashing methods into GENIE, we first propose a revised definition of the approximate nearest neighbor search, called Tolerance-Approximate Nearest Neighbor search (τ-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 [51], have used a concept similar to Definition 5.1 for ANN search though without explicit definition. The τ-ANN holds the same spirit of the definition of c-ANN.

According to the definition in [13], a hashing function $h(\cdot)$ is said to be locality sensitive if it satisfies:

$$Pr[h(p) = h(q)] = sim(p, q) \quad (1)$$

which means the collision probability is equal to the similarity measure. Another definition of the LSH is formulated that the collision probability is larger than a threshold with respect to a radius of query points. We postpone the discussion of this definition to Section 5.2.

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):

$$\hat{s} = \frac{c}{m} \quad (2)$$

Eqn. 2 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. 2 has been discussed in [51]). In fact, Eqn. 2 has been implicitly used to tune the number of LSH functions [13, 49]. 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 $s = sim(p, q)$. Next, we introduce how to integrate the LSH into our GENIE system, followed by an estimation of error bound on the $\hat{s}$ estimated by MLE.

[3] Note that the point p_i and object O_i are different concepts in this paper. We reserve the object as a concept of the match-count model. In this paper, the point presents the general searching objects/items, e.g. documents.

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. For example, the signature of one minHash function with SHA1 hashing can be hundreds of bits, and the signature of some hash function may be thousands of bits (such as the Random Binning Hashing introduced later). Meanwhile, it is not reasonable to discretize the hashing signature in to a set of buckets, since the hashing signature of two points in the same discretized bucket would have different similarity, which will validate the definition of LSH in Eqn. 1.

To tackle this problem, 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 5 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.

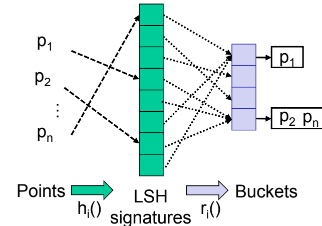


Figure 5: Re-hashing mechanism. The hashed object is $O_i = [r_1(h_1(p_i)), r_2(h_2(p_i)), ..., r_m(h_m(p_i))]$ where $h(\cdot)$ is a LSH function and $r(\cdot)$ is a random projection function.

After building the inverted index, the ANN search can be conveniently supported by GENIE. Given a query point q, we also convert the Q with the same transformation process shown in Figure 5, 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, hash function number and τ-ANN

In this section, we prove that the top return of GENIE for a query q is the τ-ANN of q. To prove this, we first analyze the error bound of the estimated similarity of Eqn. 2, whose close-form error bound is clarified in Theorem 5.1. The proof of Theorem 5.1 is inspired by the routine of the proof of Lemma 4.2.2 in [2].

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 $f_i(\cdot) = r_i(h_i(\cdot)), 1 \leq i \leq m$ with $m = 2\frac{ln(3/\delta)}{\epsilon^2}$, we can convert a point p and a query q as 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. For convenience, let c be the the return of match-count model $c = MC(O_p, Q_q)$, 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:

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

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

Number of hash functions in practical applications.

Theorem 5.1 provides a rule to set the number of LSH functions to guarantee the error bound of the estimated similarity based on our match-count model. The problem of Theorem 5.1 is that 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 here 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. As mentioned in the proof of Theorem 5.1, 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:

$$Pr[f_i(p) = f_i(q)] = Pr[r_i(h_i(p)) = r_i(h_i(q))] \tag{4}$$
$$\leq Pr[h_i(p) = h_i(q)] + Pr[r_i(h_i(p)) = r_i(h_i(q))] \tag{5}$$
$$= s + 1/D \tag{6}$$

where $s = sim(p, q)$. Thus, we have $s \leq Pr[f_i(p) = f_i(q)] \leq s + 1/D$. If we assume D is large enough, we can see $Pr[f_i(p) = f_i(q)] \approx s$. Therefore, suppose $r(\cdot)$ can re-hash $h_i(\cdot)$ into a very large domain $[0, D)$, we can claim that:

$$Pr[|MC(O_p, Q_q)/m - sim(p, q)|] \approx Pr[|c_h/m - s|] \tag{7}$$

where c_h is the collision number of LSH functions.

Eqn. 7 can be further justified by the following equation:

$$Pr[|\frac{c_h}{m} - s| \leq \epsilon] = Pr[(s - \epsilon) * m \leq c_h \leq (s + \epsilon) * m] \tag{8}$$

$$= \sum_{c_h = \lfloor (s-\epsilon)m \rfloor}^{\lceil (s+\epsilon)m \rceil} \binom{m}{c_h} s^{c_h} (1 - s)^{m - c_h} \tag{9}$$

The problem of Eqn. 9 is that the probability of error bound depends on the similarity measure $s = sim(p, q)$ [51]. Therefore, there is no closed-form expression for such error bound.

Nevertheless, Eqn. 9 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 $Pr[|c_h/m - s| \leq \epsilon] \geq 1 - \delta$ according to Eqn. 9. Figure 6 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 [51]. As we can see from Figure 6, the largest required number of hash functions 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 6 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. 9 (like Figure 6).

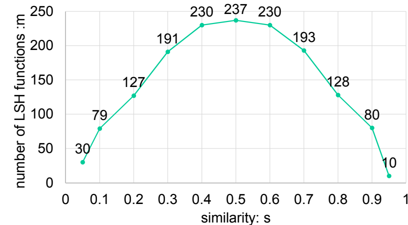


Figure 6: Similarity (s) v.s. the number of minimum required LSH functions (m) with constraint $Pr[|c_h/m - s| \leq \epsilon] \geq 1 - \delta$ **where** $\epsilon = \delta = 0.06$. **By Theorem 5.1,** $m = 2\frac{ln(3/\delta)}{\epsilon^2} = 2174$.

Tolerance Approximate NN (τ-ANN) Search.

Now we give a proof that, given a query point q, the top result returned by GENIE is 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 as 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 5) which satisfies* $|MC(O_{p_i}, Q_q)/m - sim(p_i, q)| \leq \epsilon$ *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* $sim(p^*, q) - sim(p, q) \leq 2\epsilon$ *with probability at least* $1-2\delta$.

PROOF. For convenience, we denote that the output count values of match-count model as $c = MC(O_p, Q_q)$ and $c^* = MC(O_{p^*}, Q_q)$, 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 $s^* - s \leq c^*/m + \epsilon - (c/m - \epsilon) \leq 2\epsilon$

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

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. LSH for similarity search under Eqn. 1 has been extensively studied in recent years since the pioneering work [13].

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 [48] propose a 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 is not applicable to the 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. Here we give a brief introduction about the Random Binning Hashing.

DEFINITION 5.2 (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:*

$$h(p) = [\lfloor (p^1 - u^1)/g \rfloor, ..., \lfloor (p^d - u^d)/g \rfloor] \tag{10}$$

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 [48].*

In our 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 (in the experiment, we set $D = 8192$). We select MurmurHashing3 [6] as the random projection function.

5.2 ANN search 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 [16]:

DEFINITION 5.3. *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 [16]:

$$h(q) = \lfloor \frac{\mathbf{a}^T \cdot q + b}{w} \rfloor \tag{11}$$

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 $[0, w)$.

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. 11 by the following equation [16] (let $\Delta = \parallel p - q \parallel_\mathfrak{p}$):

$$\psi_\mathfrak{p}(\Delta) = Pr[h(p) = h(q)] = \int_0^w \frac{1}{\Delta} \phi_\mathfrak{p}(\frac{t}{\Delta})(1 - \frac{t}{w})dt \tag{12}$$

where $\phi_\mathfrak{p}(\Delta)$ denotes the probability distribution density function (pdf) of the absolute value of the $\mathfrak{p}$-stable distribution. In Euclidean (l_2) space, we have $\phi_2(\Delta) = \frac{2}{\sqrt{2\pi}} exp(\frac{-\Delta^2}{2})$.

From Eqn. 12, 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 ($\Delta_1 = \parallel p_1 - q \parallel_\mathfrak{p} < \Delta_2 = \parallel p_2 - q \parallel_\mathfrak{p}$), 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.

$$sim_{l_\mathfrak{p}}(p, q) = \psi_\mathfrak{p}(\Delta) = \psi_\mathfrak{p}(\parallel p - q \parallel_\mathfrak{p}). \tag{13}$$

Now we can use GENIE to support the ANN search in high dimensional space with the help of the $(r_1, r_2, \rho_1, \rho_2)$-sensitive hashing function family. 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. 12 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. 13. Though the ANN search result is not measured by the $l_\mathfrak{p}$ norm distance, the returned results follows the same criterion to select the nearest neighbor since the similarity measure defined in Eqn.13 is closely related with the l_p norm distance.

A similar counting method has also been used for c-ANN search in [19] 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.

5.3 Searching on original data

GENIE also provides a choice of adopting the "Shotgun and Assembly" framework for similarity search. By this framework, given a dataset, we split each object of the data into small units. Then we build an 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 in the index.

The result 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 query document and the one of data 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 short document data and relational data using GENIE.

5.3.1 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. During the similarity search, GENIE can access the posting lists in parallel whose keywords are contained by the query document to calculate the match-count model between query documents and data documents. The result of the match-count model can be considered as a similarity measure of documents.

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.

5.3.2 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. Comparing the SQL *Select/Where clause* to retrieve tuples that *strictly* satisfy a set of specified criteria, GENIE can further support top-k queries to not only return the tuples satisfying all the criteria, but may also return some tuples best matching the criteria (i.e. missing the fewest criteria).

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.

6. EXPERIMENTS

6.1 Settings

6.1.1 Datasets

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

[**OCR**][4] This is a dataset from a machine learning competition for optical character recognition [54]. It contains 3,500,000 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). There is a binary label ({+1, -1}) for each point. We use RBH (see Section 5.1.3) to generate the LSH signature, which is further rehashed into an integer domain of [0,8192) (see Section 5.1.3). The prediction performance (precision, recall, F1-score and accuracy) is used to evaluate the quality of ANN search.

[**SIFT**][5] This dataset [28] contains 4,455,091 SIFT features [40] extracted from 1,491 photos, and each feature is represented as 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 [16, 3] and each function transforms a feature into 67 buckets. The setting of the bucket width follows the routine in [16], which is a trade-off between time cost and accuracy. We use this dataset to evaluate the ANN search in high dimensional space as discussed in Section 5.2.

[**Tweets**][6] This dataset has 6,770,945 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[7]. We reserve 10K tweets as a query set for the experiment. It is used to study the short document similarity search (see Section 5.3.1).

[**Adult**][8] This dataset has census information, available in the UCI Machine Learning Repository [39]. It contains 48,842 rows with 14 attributes (mixed of numerical and categorical ones). For numerical data, we discretize all value into 1024 intervals of equal width. For categorical data, we encode the original data to integers. We further duplicate every row 20 times. Thus, in total, there are 976,840 instances (after duplication). We use this dataset to study the selection from relational data (see Section 5.3.2). We select 10K tuples as queries. For numerical attributes, the query item range is defined as $[discretized_value - 50, discretized_value + 50]$; and for categorical attributes, we use exact match.

6.1.2 Competitors

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

[**GPU-LSH**] GPU-LSH is a GPU-based LSH method [46] in Euclidean space and its source code is publicly available[9]. We use GPU-LSH as a competitor of GENIE for ANN search in high dimensional space. Furthermore, since there is no

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

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

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

[7] The keywords include "Singapore", "City", "food joint", "restaurant" and "seating area", etc. This dataset is crawled for a research project.

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

[9] http://gamma.cs.unc.edu/KNN/

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 generic similarity measure. Before evaluating the running time of similarity search, we adjust the configuration parameters of GPU-LSH to make sure the ANN search results of GPU-LSH have similar quality with the one of GENIE. For convenience, we postpone the discussion about the parameter configuration of GPU-LSH in Section 6.4. The GPU-LSH cannot work with more than 1M data points since OCR data has high number of dimensions (which is ten times of SIFT data), thus, we only use 1M data points for GPU-LSH on OCR dataset.

[**GPU-Scan**] We also implemented a scanning method based 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 method to extract the top-k candidates from the array for each query. We adopt an existing k-selection method [1] on the GPU. Moreover, since the existing method only supports the k-selection for one query, in order to support parallel k-selections for multiple queries, we modify their method to use one block to handle one k-selection for one query. We give a brief introduction to the k-selection algorithm in Appendix A. 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. After finishing the scan, we use a partial quick selection function (with $\Theta(n+klogn)$ worst-case performance) in C++ STL [29] to get the k largest-count candidate objects.

[**GEN-noCH**] This is a variant of GENIE but without the Count Heap. We still build inverted index on the GPU for each dataset as we discussed in Section 4.1. However, instead of using Count Heap structure introduced in Section 4.2, we use k-selection method (which is the same with the one for GPU-Scan) to extract candidates from the Count Table. This method requires large memory requirement for multiple queries.

6.1.3 Environment

We conducted the experiment on a CPU-GPU platform. The GPU is NVIDIA GeForce GTX TITAN X with 12 GB memory. All the GPU codes were implemented with CUDA 7. All 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 $k = 100$ (i.e. to find the 100 most similar objects from dataset) 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 6.5). 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 $m = 237$ (as illustrated in Figure 6).

6.2 Search time for multiple queries

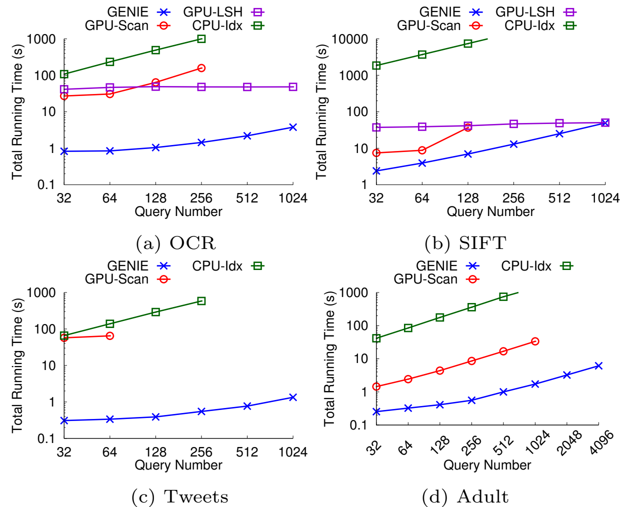


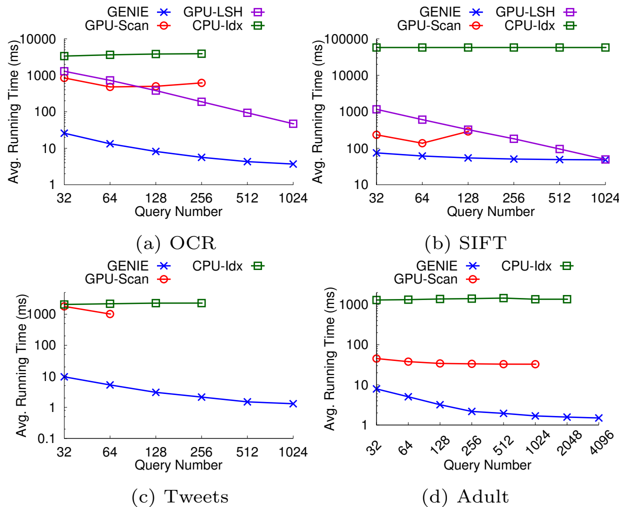
Figure 7: Total running time for multiple queries.

We show the total running time for the different numbers of queries in Figure 7 (y-axis is log-scaled). We do not show the running time for GPU-Scan 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. It is not surprising that GENIE is several orders of magnitude faster than the CPU-Idx method. Moreover, our method can also outperform the GPU-Scan method by more than one order of magnitude. For example, given 128 queries GENIE can usually finish tasks within 1 second on OCR and Tweets, but GPU-Scan needs more than one minute. Furthermore, GPU-Scan 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 7, GENIE can also achieve better (or at least not worse) 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, therefore, 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-Scan method when the query number is small since part of computational capability is 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 on the OCR dataset and has the similar running time on the SIFT dataset. Note that we only use 1M data points for GPU-LSH since it cannot work when the number of points is larger than 1M.

Figure 8 (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.

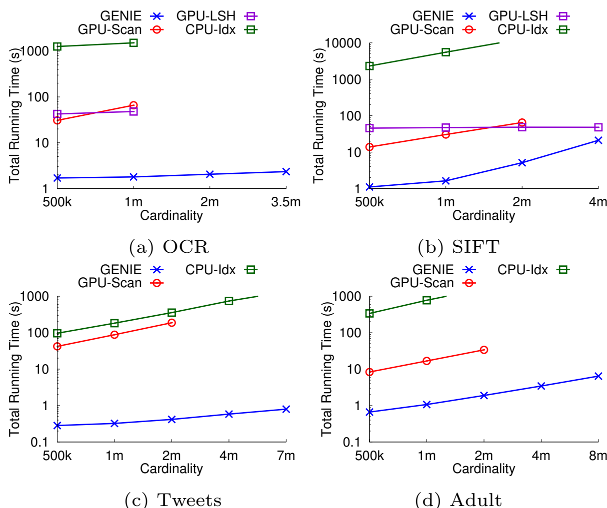
Figure 9 conveys the running time of GENIE and its competitors with varying numbers of data points from each



(a) OCR (b) SIFT

(c) Tweets (d) Adult

Figure 8: Average running time per query.



(a) OCR (b) SIFT

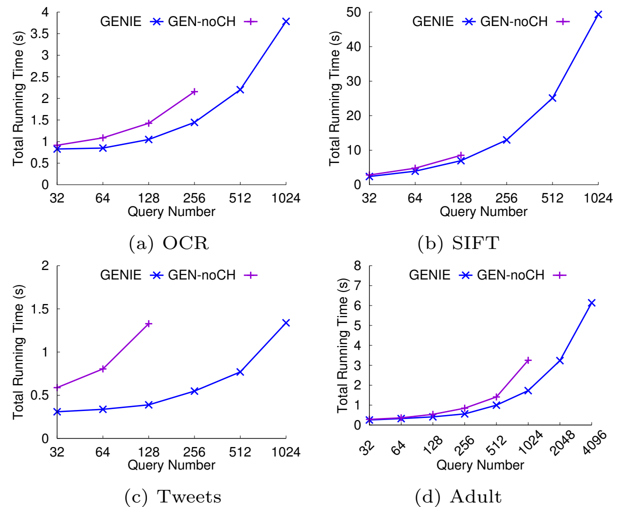
(c) Tweets (d) Adult

Figure 9: Varying data size for multiple queries. (The query number is 512 since most competitors cannot run 1024 queries for one batch.)

dataset (the Adult dataset is further duplicated to 8 million here). Since most of the competitors cannot run 1024 queries for one batch even when there are only 1M data points, we fix the query number as 512 in this figure. The running time of GENIE is gradually increased with the growth of data size. Nevertheless, the running time of GPU-LSH is relative stable 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.

6.3 Effectiveness of the Count Heap

Figure 10 demonstrates the effectiveness of the Count Heap in GENIE. The Count Heap can not only reduce the running time for similarity search given the same number of queries, but also significantly increase the number of queries to be processed by the GPU within one batch. The effectiveness of the Count Heap can be justified from two views: 1) the Count Heap can reduce the running time by avoid-



(a) OCR (b) SIFT

(c) Tweets (d) Adult

Figure 10: The effectiveness of the Count Heap.

ing selecting the candidates from a large Count Table for multiple queries. As we can see from Figure 10, when the number of queries is the same, with the help of the Count Heap, the running time of GENIE have a 2-4 fold decrease. 2) the Count Heap can also increase the maximum number of queries that can be parallel processed on the GPU within one batch, since the Count Heap can significantly reduce the memory requirement for GENIE. In Figure 10, the deployment of the Count Heap results in more than 4-fold increase of the maximum number of queries per batch on the GPU.

6.4 ANN Search of GENIE

In this section, we evaluate the quality of the ANN search of GENIE. As mentioned in Section 6.1.2, we also discuss 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. 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:

$$\frac{1}{k}\sum_{i=1}^{k}\frac{\|p_i - q\|_\mathfrak{p}}{\|p_i^* - q\|_\mathfrak{p}} \tag{14}$$

In the experiment evaluation (especially for running time) for ANN search in high dimensional space on the SIFT data 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 6). Another parameter for ANN search in high dimensional space is the bucket width (of Eqn. 11). According to the method discussed in the original paper of E2LSH [16], 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

Table 1: 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

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.

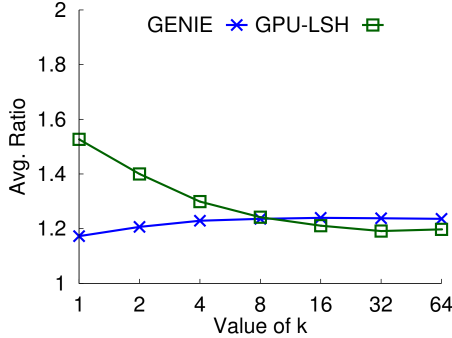


Figure 11: **Approximation ratio v.s. value of k on SIFT data**

Figure 11 shows the approximation ratio of GPU-LSH and GENIE with varying the value of k. From Figure 11, 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. In any way, 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. 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]. 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 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 1 where the number of hash tables for GPU-LSH is set as 100. Note that the prediction

Table 2: Time profiling of different stages of GENIE for 1024 queries (the unit of time is $millionsecond$).

Stage		OCR	SIFT	Tweets	Adult
Index building		78082.12	115273.71	10999.25	946.05
Query	transfer	272.06	291.03	8.77	12.16
	match	3209.95	49016.56	1137.24	1509.95
	select	27.02	24.33	18.60	11.71

performance of GPU-LSH is slightly worse then the one of GENIE. It is possible to improve the performance of GPU-LSH by increasing the number of hash functions, which will dramatically increase the running time for queries.

6.5 Experimental study on load balance

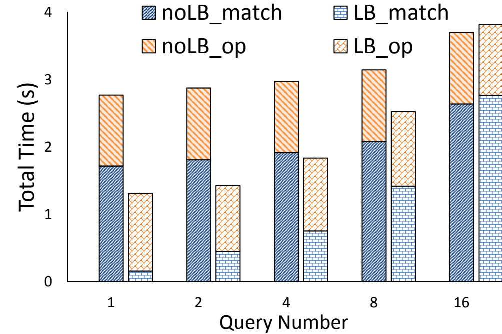


Figure 12: **Load balance on Adult data (with 100M data points)**

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 12 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 12, 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.

6.6 Time profiling

Table 2 shows the time cost of different stages of GENIE. The "Index building" represents the running time to build the inverted index on the GPU. This is an one-time cost, and

we do not count it into the query time. The rows of "Query" show 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 the Count Heap and sending back the candidates to the CPU memory. 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 list scanning operation. This justifies the rationality for developing an inverted index on the GPU.

7. CONCLUSION

In this paper, we presented GENIE, a generic inverted index for multiple similarity search queries on the GPU. GENIE can support the tolerance-Approximate Nearest Neighbour (τ-ANN) search for various data types with any similarity measure satisfying a generic Locality Sensitive Hashing scheme, as well as similarity search on original data within a "Shotgun and Assembly" framework. In order to process more queries simultaneously in a batch on the GPU, we proposed a novel data structure – Count Heap, which reduces the memory requirement significantly for multi-query processing. In particular, in the experiment we showed how to use GENIE to support ANN search in kernel space and in high dimensional space, similarity search on short document data, and 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.

8. ACKNOWLEDGMENT

This research was carried out at the NUS-ZJU Sensor-Enhanced social Media (SeSaMe) Centre. It is supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the Interactive Digital Media Programme Office.

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] I. Buck. Nvidia next-gen pascal gpu architecture to provide 10x speedup for deep learning apps. `http://blogs.nvidia.com/blog/2015/03/17/pascal/`, 2015.

[11] 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.

[12] P. Celis, P.-Å. Larson, and I. J. Munro. Robin hood hashing. In *FOCS*, pages 281–288, 1985.

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

[14] 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.

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

[16] 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.

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

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

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

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

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

[22] 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.

[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] N. M. Josuttis. *The C++ standard library: a tutorial and reference*. Addison-Wesley, 2012.

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

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

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

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

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

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

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

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

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

[47] 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.

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

[49] 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.

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

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

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

[53] 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.

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

[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] 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.

[66] 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.

[67] 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.

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

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

APPENDIX

A. K-SELECTION ON THE GPU

To extract the top-k object from an array, we modify a GPU-based bucket-selection algorithm [1] for this purpose. Figure 13 shows an example for such selection process.

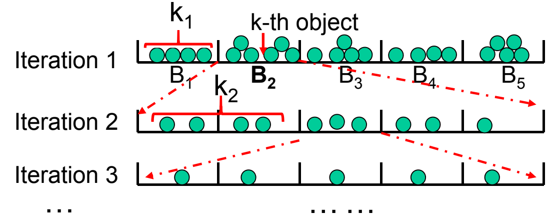


Figure 13: Example for bucket k-selection. The iteration repeats until $k = k_1 + k_2 + ... + k_t$.

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 13, 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 13, 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.