A Generic Inverted Index Framework for Similarity Search on the GPU

Jingbo Zhou[†*], Qi Guo[†], H. V. Jagadish[#], Luboš Krčál[†], Siyuan Liu[‡]
Wenhao Luan[†], Anthony K. H. Tung[†], Yueji Yang[†§], Yuxin Zheng[†§]
[†]National University of Singapore [#]Univ. of Michigan, Ann Arbor [‡]Nanyang Technological University
[*]Business Intelligence Lab, Baidu Research [§]Tencent Inc.
[†]{jzhou, qiguo, krcal, luan1, atung, yueji, yuxin}@comp.nus.edu.sg
[#]jag@umich.edu, [§]sliu019@e.ntu.edu.sg

***Abstract*—We propose a novel generic inverted index framework on the GPU (called GENIE), aiming to reduce the programming complexity of the GPU for parallel similarity search of different data types. Not every data type and similarity measure are supported by GENIE, but many popular ones are. We present the system design of GENIE, and demonstrate similarity search with GENIE on several data types along with a theoretical analysis of search results. A new concept of locality sensitive hashing (LSH) named τ-ANN search, and a novel data structure c-PQ on the GPU are also proposed for achieving this purpose. Extensive experiments on different real-life datasets demonstrate the efficiency and effectiveness of our framework. The implemented system has been released as open source[1].**

I. Introduction

There is often a need to support a high throughput of queries on index structures at scale. These queries could arise from a multiplicity of "users", both humans and client applications. Even a single application could sometimes issue a large number of queries. For example, image matching is often done by extracting hundreds of high dimensional SIFT (scale-invariant feature transform) features and matching them against SIFT features in the database. Parallelization for similarity search is required for high performance on modern hardware architectures [1], [2], [3], [4].

Solutions may be implemented it on Graphics Processing Units (GPUs). GPUs have experienced a tremendous growth in terms of computational power and memory capacity in recent years. One 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. Furthermore, most PCs allow two to four GPUs to be installed, bringing the total amount of GPU memory in a PC to be comparable with a regular CPU memory.

However, GPU programming is not easy. Effectively exploiting parallelism is even harder, particularly as we worry about the unique features of the GPU including the Single-Instruction-Multiple-Data (SIMD) architecture, concurrent control, coherent branching and coalescing memory access. While capable programmers could take their index structure of choice and create a GPU-based parallel implementation, doing so will require considerable effort and skill.

Our goal is to address this parallel similarity search problem on the GPU in a generic fashion. To this end, we develop an efficient and parallelizable GPU-based Generic Inverted Index framework, called GENIE (we also name our system as GENIE), for similarity search using an abstract *match-count model* we define. GENIE is designed to support parallel computation of the match-count model, but the system is generic in that a wide variety of data types and similarity measures can be supported.

We do not claim that every data type and similarity measure are supported by GENIE[2] – just that many are, as we will demonstrate in the paper, including most that we have come across in practice. As an analogy, consider the map-reduce model, implemented in a software package like Hadoop. Not all computations can be expressed in the map-reduce model, but many can. For those that can, Hadoop takes care of parallelism and scaling out, greatly reducing the programmer's burden. In a similar way, our system, GENIE, can absorb the burden of parallel GPU-based implementation of similarity search methods and index structures.

Our proposed match-count model defines the common operations on the generic inverted index framework which has enough flexibility to be instantiated for different data types. The insight for this possibility is that many data types can be transformed into a form that can be searched by an inverted-index-like structure. Such transformation can be done by the *Locality Sensitive Hashing* (LSH) [5], [6] scheme under several similarity measures or by the *Shotgun and Assembly* (SA) [7], [8] scheme for complex structured data. We present a detailed discussion of this in Section II-B.

The first challenge of GENIE is to design an efficient index architecture for the match-count model. We propose an inverted index structure on the GPU which can divide the query processing to many small tasks to work in a fine-grained manner to fully utilize GPU's parallel computation power.

[‡]Siyuan Liu has done his work on the project as an intern at NUS.
[1]https://github.com/SeSaMe-NUS/genie

[2]Note that we named our system as "generic inverted index", but not "general inverted index".

GENIE also exploits GPU's properties like coalescing memory access and coherence branching during the index scanning.

We propose a novel data structure on the GPU, called Count Priority Queue (c-PQ for short), which can significantly reduce the time cost for similarity search. Due to the SIMD architecture, another challenge of GENIE is how to select the top-k candidates from the candidate set, which is widely considered a main bottleneck for similarity search on the GPU in previous study [9], [4] (which is called k-selection in [9] and short-list search in [4]). Existing methods usually adopt a sorting method which is an expensive operation, or a priority queue on the GPU which have warp divergence problem and irregular memory movement. Our novel design of c-PQ can keep only a few candidates on a hash table on the GPU, and we only need to scan the hash table once to obtain the query result. Therefore this major bottleneck for similarity search on the GPU can be overcome.

We optimize data placement on the GPU to improve the throughput of GENIE. The novel structure of c-PQ can also reduce the memory requirement for multiple queries. Therefore GENIE can substantially increase the number of queries within a batch on the GPU. We propose a tailored hash table on the GPU to reduce hash confliction. Besides, to overcome the limited memory size of the GPU, we introduce a multiple loading strategy to process large data.

We describe how to process data using an LSH scheme by GENIE. We propose a new concept, called Tolerance-Approximate Nearest Neighbour (τ-ANN) search, which is in the same spirit as the popular c-ANN search. Then we prove that, GENIE can support the τ-ANN search for any similarity measure that has a generic LSH scheme.

For complex data types without LSH transformation, another choice is to adopt the SA scheme to process the data. We will showcase this by performing similarity search on sequence data, short document data and relational data using GENIE.

We summarize our contributions as follows:

- We propose a generic inverted index framework (GENIE) on the GPU, which can absorb the burden of parallel GPU-based implementation of similarity search for any data type that can be expressed in the match-count model.
- We present the system design of GENIE. Especially, we devise the novel data structure c-PQ to significantly increase the throughput for query processing on the GPU.
- We exhibit an approach to adopting LSH scheme for similarity search under GENIE. We propose the new concept of τ-ANN, and demonstrate that GENIE can effectively support τ-ANN search under the LSH scheme.
- We showcase the similarity search on complex data structures by GENIE under the SA scheme.
- We conduct comprehensive experiments on different types of real-life datasets to demonstrate the effectiveness and efficiency of GENIE.

We enclose all proofs and supplementary materials in our technique report [10].

II. Preliminaries and Overview

In this section, we give an overview of GENIE including its main concepts and computational framework. We use relational data shown in Fig. 1 as a running example.

A. Match-count model

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 $QS = \{Q_1, ..., Q_m\}$.

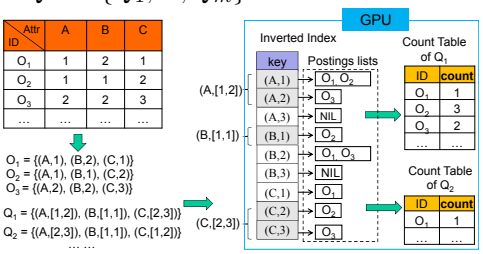


Fig. 1. An example on a relational table.

Example 2.1: Given a relational table, the universe U is a set of ordered pairs (d, v) *where d is an attribute of this table and v is a value of this attribute. An $l-$dimensional relational tuple $p = (v_1, ..., v_l)$ is represented as an object $O = \{(d_1, v_1), ..., (d_l, v_l)\}$. As illustrated in Fig. 1, the O_1 in the relational table is represented as $O_1 = \{(A, 1), (B, 2), (C, 1)\}$.*

A query on the relational table usually defines a set of ranges R=$([v_1^L, v_1^U]$,..., $[v_l^L, v_l^U])$. Then it can be represented as Q=$\{r_1, r_2, .., r_l\}$, where $r_i = (d_i, [v_i^L, v_i^U])$ defines a set of pairs (d_i, v) with value $v \in [v_i^L, v_i^U]$. As we can see from Fig. 1, query Q_1 to retrieve the tuples with conditions $1 \leq A \leq 2, 1 \leq B \leq 1$ and $2 \leq C \leq 3$ can be represented as $Q_1 = \{(A, [1, 2]), (B, [1, 1]), (C, [2, 3])\}$.

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

Definition 2.1 (match-count model): Given a query $Q = \{r_1, r_2, .., r_l\}$ and an object $O = \{o_1, ..., o_s\}$, we map each query item r_i to a natural integer using $C : (r_i, O) \to \mathbb{N}$, where $C(r_i, O)$ returns the number of elements $o_j \in O$ contained by the item r_i (which is also a subset of U). Finally the output of the match-count model is the sum of the integers $MC(Q, O) = \sum_{r_i \in Q} C(r_i, O)$. For example, in Fig. 1, for Q_1 and O_1 we have $C((A, [1, 2]), O_1) = 1, C((B, [1, 1]), O_1) = 0$ and $C((C, [2, 3]), O_1) = 0$, then we have $MC(Q_1, O_1) = 1 + 0 + 0 = 1$.

In GENIE, we aim to rank all the objects in a data set with respect to the query Q according to the model $MC(\cdot, \cdot)$ to obtain the top-k objects of query Q.

GENIE essentially is an inverted index on the GPU to efficiently support the match-count model between objects and queries. Fig. 1 shows an illustration of such high level inverted index. 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 *postings 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 postings lists, we can calculate the match counts between the query and all objects.

B. GENIE with LSH and SA

The inverted index with match-count model has the flexibility to support similarity search of many data types. Just as map-reduce model cannot handle all computation tasks, we do not expect that all data types can be supported. However, at least many popular data types can be supported with LSH or SA as we address further below. We illustrate the relationships among GENIE, LSH and SA in Fig. 2. How to organize data structures as inverted indexes has been extensively investigated by previous literature [2], [11], [12] and it is beyond the scope of this paper.

1) Transformed by LSH: The most common data type, high dimensional point, can be transformed by an LSH scheme [6]. In such a scheme, multiple hash functions are used to hash data points into different buckets and points that are frequently hashed to the same bucket are deemed to be similar. Hence we can build an inverted index where each postings list corresponds to a list of points hashed to a particular bucket. Given a query point, ANN search can be performed by first hashing the query point and then scanning the corresponding postings list to retrieve data points that appear in many of these buckets. Meanwhile, sets, feature sketches and geometries typically have kernelized similarity functions [5], including Jaccard kernel for sets, Radial Basis Function (RBF) kernel for feature sketches, and Geodesic kernel for hyperplanes. We present the index building method under LSH scheme and theoretical analysis in Section IV.

2) Transformed by SA: The data with complex structure, including documents, sequences, trees and graphs, can be transformed with the SA [7], [8] scheme. Specifically, the data will be broken down into smaller sub-units ("shotgun"), such as words for documents, n-grams for sequences [2], binary branches for trees [12] and stars for graph [11]. After the decomposition, we can build an inverted index with a postings list for each unique sub-unit. Data objects containing a particular sub-unit are stored in the postings list. At query time, query objects will also be broken down into a set of small sub-units and the corresponding postings lists will be accessed to find data objects that share common sub-units with the query object. The match-count model returns the matching result between the query and objects sharing keywords, which is an important intermediate result for similarity search. This approach has been widely used for similarity search of complex structured data [2], [12], [11]. More discussion about it is presented in Section V.

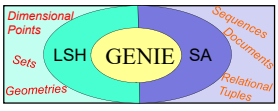


Fig. 2. The relations among GENIE, LSH and SA.

III. Inverted Index on the GPU

We first present the index structure and the data flow of GENIE. Then we present Count Priority Queue (c-PQ for short), which is a priority queue-like structure on the GPU memory facilitating the search. Finally, we propose a multiple loading method to handle large dataset.

A. Inverted index and query processing

The inverted index is resident in the global memory of the GPU. Fig. 3 illustrates an overview of such an index structure. All postings lists are stored in a large *List Array* in the GPU's global memory. There is also a *Position Map* in the CPU memory which stores starting and ending positions of each postings list for each keyword in the List Array. When processing queries, we use the Position Map to look up the corresponding postings list address for each keyword. This look-up operation is only required once for each query and our experiment also demonstrates that its time cost is negligible.

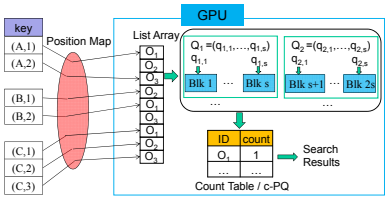


Fig. 3. Overview of the inverted index and data flow.

Fig. 3 shows the process of multiple queries on the GPU. Each query has a set of items which define particular ranges on some attributes. When we invoke a query, we first obtain its postings lists' addresses by the Position Map, then we use one block of the GPU (A block on the GPU organizes a small batch of threads (up to 2048) and controls the cooperation among the threads.) to scan the corresponding postings lists for each query item, where the threads of each block parallel access parts of the postings lists. For a query $Q_i = \{q_{i,1}, q_{i,2}, ..., q_{i,s}\}$ with s query items, if there are m queries, there will be about $m \cdot s$ blocks working on the GPU in parallel. During the process, after scanning an object in the postings list, each thread will update the *Count Table* to update the number of occurrences of the object in the scanned postings lists. Therefore, the system works 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 extremely long postings lists, which can be a bottleneck for our system. We also consider how to balance the workload for each block by breaking long postings lists into short sub-lists. We refer readers to [10] for more details about load balancing.

B. Count Priority Queue

We propose a novel data structure, called Count Priority Queue (c-PQ for short) to replace the Count Table on the GPU, which aims to improve the efficiency and throughput of GENIE. c-PQ has two strong points: 1) Though how to retrieve the top-k result from all candidates is the major bottleneck for similarity search on the GPU [4], c-PQ can finish this task with small cost; and 2) c-PQ can significantly reduce the space requirement of GENIE.

One major obstacle of GENIE is how to select top-k count objects from the Count Table. This problem is also considered a major bottleneck for similarity search on the GPU in previous study [4]. It is desirable to use a priority queue for this problem. However, the parallel priority queue usually has warp divergence problem and irregular memory movement, which cannot run efficiently on GPU architectures [13].

The key idea of c-PQ is to use a two-level data structure to store the count results, and use a device to schedule the data allocation between levels. Our novel design can guarantee that only a few of candidates are stored in the upper level structure while all objects in lower level structure can be abandoned. Then we only need to scan the upper level structure to select the query result. We will describe the structure and mechanism of c-PQ, and prove all the claims in Theorem 3.1.

Another major problem of GENIE is its large space cost, since the Count Table must allocate 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.

To reduce space cost, first, we can use bitmap structure to avoid explicitly storing id. Second, we only need to allocate several (instead of 32) bits to encode the count for each object in bitmap structure. The reason is that the maximum count for each object is bounded (i.e. there is a maximum value of the count) since the count value cannot be larger than the number of postings lists in the index. Actually, we usually can infer a much smaller count bound than the number of postings lists. For example, for high dimensional points, the maximum count value is just the number of its dimensions.

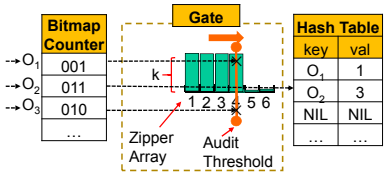


Fig. 4. An illustration of the c-PQ.

1) The structure and mechanism of c-PQ: Fig. 4 shows the main structures of c-PQ which has three components. In the lower level, we create a *Bitmap Counter* which allocates several bits (up to 32 bits) for each objects whose id is corresponding to the beginning address of the bits. In the upper level, there is a *Hash Table* whose entry is a pair of object id and its count value. Then, a pivotal device, called *Gate*, determines which id-value pair in the Bitmap Counter will be inserted into the Hash Table. The Gate has two members: a *ZipperArray* and a threshold called *AuditThreshold*. In the following context, we briefly denote the Bitmap Counter as BC, the Hash Table as HT, the AuditThreshold as AT and the ZipperArray as ZA.

The Gate has two functions. First, only a few objects in the BC can pass the Gate to the HT, while all objects remaining in the BC cannot be top k objects and thus can be safely abandoned. Second, the AT in the Gate just keeps track of the threshold for the top-k result, and we only need to scan the HT once to select the objects with counter larger than $AT-1$ as top-k results (See Theorem 3.1).

The ZA and AT in the Gate work together to restrict objects going from the BC to the HT. The size of the *ZA* in Gate is equal to the maximum value of the count (based on the count value bound). *ZA[i]*(ZA is 1-based indexing array, i.e. the index starts from 1) records the minimum value between the number of objects whose count have reached i (denoted as zc_i) and value k, i.e. $ZA[i] = min(zc_i, k)$. The *AT* in Gate records the minimum index of ZA whose value is smaller than k (i.e. $ZA[AT] < k$ and $ZA[AT-1] \geq k$).

The intuition behind the mechanism of the Gate is that, if there are already k objects whose count has reached i (i.e. $ZA[i] == k$) in the HT, there is no need to insert more objects whose count is less or equal to i into the HT since there are already k candidates if the top-k count threshold is just i. Therefore, the AT increase by 1 when $ZA[AT] == k$.

We present the update process per thread on the GPU of c-PQ in Algorithm 1, which is also illustrated in Fig. 4. For each query, we use one block of the GPU to parallel process one query item. For all inverted lists matched by a query item, the threads of this block access the objects and update c-PQ with Algorithm 1. Note that the add operation is atomic. When the count of an object is updated in the BC, we immediately check whether the object's count is larger than the *AT* (line 3). If it is, we will insert (or update) an entry into the HT whose key is the object id and whose value is the object's count. Meanwhile, we will update the *ZA* (line 5). If *ZA[AT]* is larger than k, we also increase the *AT* by one unit (line 7).

Algorithm 1: Update on the Count Priority Queue

```
   // For a thread in a block, it accesses
      object O_i in the inverted index, then
      makes following updates.
1  val_i = BC[O_i] + 1
2  BC[O_i] = val_i
3  if val_i ≥ AT then
4      Put entry (O_i, val_i) into the HT
5      ZA[val_i] += 1
6      while ZA[AT] ≥ k do
7          AT += 1
```

We present Theorem 3.1 to elaborate the properties of c-PQ.

Theorem 3.1: After finishing scanning the inverted index and updating c-PQ, the top-k candidates are stored in the HT, and the number of objects in the HT is $O(k * AT)$. 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 $MC_k = AT - 1$.

According to Theorem 3.1, we can select the top-k objects by scanning the HT and selecting objects with match count greater than $(AT-1)$ only. If there are multiple objects with match count equal to $(AT-1)$, we break ties randomly.

We give an example to show update process of c-PQ with data in Fig. 1 and the final result shown in Fig. 4.

Example 3.1: Given a data set $\{O_1, O_2, O_3\}$ and a query Q_1 in Fig. 1, we want to find the top-1 result of Q_1 from the

objects, i.e. $k = 1$. *Since the number of attributes of the table is* 3, *the maximum value of count is* 3. *Initially we have* $AT = 1$, $ZA = [0, 0, 0]$, $BC = \{O_1 : 0, O_2 : 0, O_3 : 0\}$ *and* $HT = \emptyset$. *For easy explanation, we assume the postings lists matched by* Q_1 *are scanned with the order of* $(A, [1, 2])$, $(B, [1, 1])$ *and* $(C, [2, 3])$. *(On the GPU they are processed with multiple blocks in parallel with random order.)*

As shown in Algorithm 1, when scanning the postings list $(A, [1, 2])$, *we first access* O_1 *and get* $BC(O_1) = 1$. *Since* $BC(O_1) \geq AT(= 1)$, *we have* $HT(O_1) = 1$ *and* $ZA[1] = 0 + 1 = 1$ *(note that ZA is 1-based array, thus after the updating* $ZA = [1, 0, 0]$*). Since* $ZA[AT] \geq k$ *(*$k = 1$ *and* $AT = 1$*), then we have* $AT = 1 + 1 = 2$. *Then we update* $BC(O_2) = 1$ *and* $BC(O_3) = 1$ *without changing other parameters since both the values of* O_2 *and* O_3 *are smaller than* AT.

With the same method, after processing $(B, [1, 1])$, *we only have* $BC(O_2) = 2$, $HT(O_2) = 2$, $ZA = [1, 1, 0]$, $AT = 3$ *and* $BC = \{O_1 : 1, O_2 : 2, O_3 : 1\}$.

The last postings list to process is $(C, [2, 3])$. *There is no* O_1 *in it. For* O_2, *we have* $BC(O_2) = 3$. *Since* $BC(O_2) \geq AT$, *we have* $HT(O_2) = 3$, $ZA = [1, 1, 1]$ *and* $AT = 4$. *We also have* $BC(O_3) = 2$.

Finally, we have $HT = \{O_1 : 1, O_2 : 3\}$ *and* $AT = 4$. *By Theorem 3.1, we know that the count of top-1 result is 3 (AT-1=4-1=3). We can then scan the hash table* HT *to select the object equal to* 3 *which is just* O_2.

2) Hash Table with modified Robin Hood Scheme: Here we briefly explain the design of the HT. We propose a modified Robin Hood Scheme to implement a hash table on the GPU which is different from existing work [14]. According to Theorem 3.1, the size of the HT can be set as $O(k * max\_count\_value)$. We adopt a lock-free synchronization mechanism studied in [15] to handle the race condition problem. More details about the HT can be found in [10].

The vital insight to improve the efficiency of the Robin Hood Scheme in the c-PQ is that all entries with values smaller than $(AT - 1)$ in the HT cannot be top-k candidates (see Theorem 3.1). If the value of an entry is smaller than $(AT - 1)$, we can directly overwrite the entry regardless of hashing confliction. Thus we can significantly reduce the probe times of insertion of the HT, since many inserted keys become expired with the increase of *AT*.

C. Indexing large data with multiple loadings

We also devise a multiple loading method to increase the capacity of GENIE utilizing the advantage of the high GPU memory bandwidth. There may be some cases that the data index is too large to be fitted into the GPU memory. For this problem, we split the whole data set into several parts, and then build inverted index for each part in the CPU memory. When a batch of queries is submitted, we transfer each index part into the GPU memory in turn, and run the query processing introduced before. After finishing a round, we collect all results from each part, and merge them to get a final query results. Some necessary computation is done in the CPU such as finding the final top-k results among the top-k results of each data part. Fig. 5 illustrates this multiple loading method.

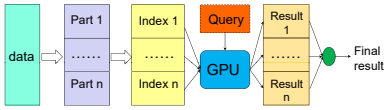


Fig. 5. An illustration of GENIE with multiple loadings.

D. The utility of the GPU for GENIE

The design of GENIE utilizes the property of the SIMD architecture and the features of the GPU from several perspectives. First, GENIE divides the query process to sufficient number of small tasks to fully utilizes the parallel computation power of the GPU. We use one block to handle one query item and use each thread of the block to process an object in the postings list, therefore, the similarity search can be processed in as fine-grained manner as possible, so that all the processors on the GPU are highly utilized.

Second, c-PQ can finish the top-k selection task with a small number of homogenous operations (scanning the Hash Table only once) suitable for the SIMD architecture. This data structure avoids the expensive operations like sort [4], [9] and data dependent memory assess movement on the GPU [13] for top-k selection. This point is clarified in the experiment of Section VI-B1 and Section VI-C, with a discussion about its competitors in Section VI-B4.

Third, the design of GENIE tries to perform coalescing memory access and coherent branching. In c-PQ, most operations are limited in the BC and only a few data are passed to the HT, which minimizes the branch divergence. Besides, since we use many threads to process a postings list, the threads have coalescing memory access patterns.

Fourth, the multiple loading method takes the advantage of the high GPU memory bandwidth which is usually 5-10 times higher than the CPU memory bandwidth. Our experiment in Section VI-B3 also demonstrates that such index transfer step only takes a very small portion of the total time cost.

IV. Generic ANN Search with LSH

We first show that GENIE can support the ANN search for any similarity measure which has an LSH scheme, followed by an error bound analysis for ANN search on GENIE.

A. Building index for ANN search on GENIE

In this section, we show how to use GENIE to support similarity search after processing data by LSH scheme.

1) ANN search with LSH on GENIE: According to the definition in [5], a hashing function $h(\cdot)$ is said to be locality sensitive if it satisfies:

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

which means the collision probability is equal to the similarity measure. Here $sim(\cdot, \cdot)$ is a function that maps a pair of points to a number in $[0, 1]$ where $sim(p, q) = 1$ means p and q are identical. LSH is one of the most popular solutions for the ANN search problem [16], [5], [17].

We can use the indexing method for relation table shown in Fig. 1 to build inverted index for LSH. 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 $(i, h_i(p))$ and the postings list of the pair $(i, h_i(p))$ is a set of points whose hash value by $h_i(\cdot)$ is $h_i(p)$ (i.e. $h_i(p') = h_i(p)$ if p' and p in the same postings list.). Given a query point q, we also convert q with the same transformation process, i.e. $Q = [h_1(q), h_2(q), ..., h_m(q)]$.

As we will prove in Section IV-B, the top result returned by GENIE according to the match-count model on the inverted index is just the ANN search result. Any similarity measure associated with an LSH family defined by Eqn. 1 can be supported by GENIE. For ANN search in high dimensional space, we usually resort to $(r_1, r_2, \rho_1, \rho_2)$-sensitive hashing function family. We give a special discussion about it in [10]. We will also analyze the error bound between the estimate $\hat{s}$ and the real similarity measure $s = sim(p, q)$ in Section IV-B.

2) Re-hashing for LSH with large signature space: A possible problem is that the hash signature of LSH functions may have a huge number of values with acceptable error by configuring parameters. For example, the signature of the Random Binning Hashing function introduced later can be thousands of bits by setting good parameters for search error. But it is not reasonable to discretize the hash signature into a set of buckets according to the definition of LSH in Eqn. 1.

To reduce the number of possible signatures, we propose a re-hashing mechanism illustrated in Fig. 6. After obtaining the LSH signature $h_i(\cdot)$, we further project the signatures into a small set of buckets with a random projection function $r_i(\cdot)$. Thus, we can convert a point to an object by the transformation: $O_i = [r_1(h_1(p_i)), r_2(h_2(p_i)), ..., r_m(h_m(p_i))]$. Note that re-hashing is not necessary if the signature space of selected LSH can be configured small enough.

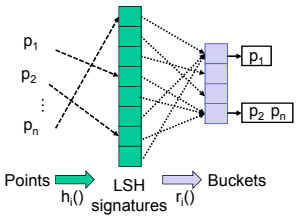


Fig. 6. Re-hashing mechanism where $h(\cdot)$ is a LSH function and $r(\cdot)$ is a random projection function.

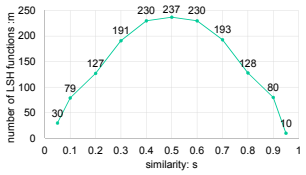


Fig. 7. Similarity (s) v.s. the number of minimum required LSH functions (m) with constraint $Pr[|c/m - s| \leq \epsilon] \geq 1 - \delta$ where $\epsilon = \delta = 0.06$.

3) Case study: ANN in Laplacian kernel space: We take the ANN search on a shift-invariant kernel space as a case study, which has important applications for machine learning. The authors in [18] propose an LSH family, called Random Binning Hashing (RBH), for Laplacian kernel $k(p, q) = exp(- \parallel p - q \parallel_1 / \sigma)$. Though this method is well-known for dimension reduction, as far as we know, it has not been applied to ANN search. One possible reason is that it has a huge hash signature space. A brief introduction to RBH can be found in [10]. In experiment, we demonstrate that GENIE can support ANN search in Laplacian kernel space based on RBH. To reduce the hash signature space, we use the re-hashing mechanism to project each signature into a finite set of buckets.

B. Theoretical analysis

To integrate LSH methods into GENIE requires a theoretical analysis for LSH under match-count model. For this purpose, we propose a revised definition of the ANN search, called Tolerance-Approximate Nearest Neighbor search (τ-ANN).

Definition 4.1 (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 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.

This concept is similar to the popular definition of c-ANN [16] which is defined to find a point p so that $sim(p, q) \leq c \cdot sim(p^*, q)$ with high probability. Some existing works, like [19], have also used a concept similar to Definition 4.1 though without explicit definition.

1) Error bound and τ-ANN: We prove that the top return of GENIE for a query q is the τ-ANN of q. Given a point p and a query q with a set of LSH functions $\mathbb{H} = \{h_1, h_2, ..., h_m\}$, suppose there are c functions in $\mathbb{H}$ satisfying $h_i(p) = h_i(q)$ (where c is just the return of match-count model). We prove in Theorem 4.1 that the return of match-count model on GENIE can be probabilistically bounded w.r.t the similarity between p and q, i.e. $|c/m - sim(p, q)| < \epsilon$ with high probability.

Theorem 4.1: Given a similarity measure $sim(\cdot, \cdot)$, an LSH family $h(\cdot)$, 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 to an object and a query of the match-count model, which are $O_p = [f_1(p), f_2(p), ..., f_m(p)]$ and $Q_q = [f_1(q), f_2(q), ..., f_m(q)]$, then we have $|MC(Q_q, O_p)/m - sim(p, q)| < \epsilon + 1/D$ with probability at least $1 - \delta$.

Now we introduce an important theorem which claims that, given a query point q and proper configuration of m stated in Theorem 4.1, the top result returned by GENIE is just the τ-ANN of q.

Theorem 4.2: Given a query q and a set of points $P = \{p_1, p_2, .., p_n\}$, we can convert them to the objects of our match-count model by transformation $O_{p_i} = [r_1(h_1(p_i)), r_2(h_2(p_i)), ..., r_m(h_m(p_i))]$ which satisfies $|MC(Q_q, O_{p_i})/m - sim(p_i, q)| \leq \epsilon$ 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 $1 - 2\delta$.

2) Number of hash functions in practice: Theorem 4.1 provides a rule to set the number of LSH functions as $O(\frac{1}{\epsilon^2})$ 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 simple tasks. The question however is that: Do we really need such a large number of hash functions in practical applications?

Before exploiting this, we first explain that the collision probability of a hash function $f_i(\cdot)$ can be approximated with the collision probability of an LSH function $h_i(\cdot)$ if D is large enough. The collision probability of $f_i(\cdot)$ can be factorized as

collisions caused by $h_i(\cdot)$ and collisions caused by $r_i(\cdot)$:

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

where $s = sim(p, q)$. Thus, we have $s \leq Pr[f_i(p) = f_i(q)] \leq s + 1/D$. Suppose $r(\cdot)$ can re-hash $h_i(\cdot)$ into a very large domain $[0, D)$, we can claim that $Pr[f_i(p) = f_i(q)] \approx s$. For simplicity, let us denote $c = MC(Q_q, O_p)$. An estimation of s by maximum likelihood estimation (MLE) can be [19]:

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

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

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

Eqn. 7 shows that the probability of error bound depends on the similarity measure $s = sim(p, q)$ [19]. Therefore, there is no closed-form expression for such error bound.

Nevertheless, Eqn. 7 provides a practical solution to estimate a tighter error bound of the match-count model. If we fixed ϵ and δ, for a similarity measure s, we can infer the number of required hash functions m subject to the constraint $Pr[|c/m - s| \leq \epsilon] \geq 1 - \delta$ according to Eqn. 7. Fig. 7 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 [19]. Fig. 7 shows that the largest number of hash functions, being m=237, appears at $s = 0.5$, which is much smaller than the one estimated by Theorem 4.1 (which is $m = 2\frac{ln(3/\delta)}{\epsilon^2} = 2174$). We should note that the result shown in Fig. 7 is data independent. Thus, instead of using Theorem 4.1, we can effectively estimate the actually required number of LSH functions using the simulation result based on Eqn. 7 (like Fig. 7).

V. Searching on data with SA

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

The return of match-count model can either be a similarity measure (e.g. document search where the count is just the inner product between the space vector of documents), or be considered as a lower bound of a distance (e.g. edit distance) to filter candidates [2], [12]. We will demonstrate how to perform similarity search on sequence data, short document data and relational data using GENIE.

A. Searching on sequence data

In this section, we show how to use GENIE to support similarity search by the SA scheme with an example of sequence similarity search under edit distance.

1) Shotgun – decomposition and index: We first decompose the sequence S into a set of n-grams using a length-n sliding window. Given a sequence S and an integer n, the n-gram is a length-n subsequence s of S. Since the same n-gram may appear multiple times in a sequence, we introduce the *ordered n-gram*, which is a pair (n-gram,i) where i denotes the i-th same n-gram in the sequence. Therefore, we decompose the sequence S into a set of ordered n-gram $G(S)$. In GENIE, we build an inverted index by treating the ordered n-gram as a keyword and putting its sequence id in the postings list.

Example 5.1: For a sequence $S = \{aabaab\}$, the set of ordered 3-grams *of S is $G(S) = \{(aab, 0), (aba, 0), (baa, 0), (aab, 1)\}$ where $(aab, 0)$ denotes the first subsequence aab in S, and $(aab, 1)$ denotes the second subsequence aab in S.*

2) Assembly – combination and verification: During query process, we also decompose a query sequence Q into a set of *ordered n-grams* using sliding windows, i.e. $G(Q)$. GENIE can retrieve candidates with top-k large count in the index. We first introduce the following lemma:

Lemma 5.1: Suppose the same n-gram s_n^i appears c_s^i times in sequence S and c_q^i times in sequence Q, then the result returned by the match-count model is $MC(G(S), G(Q)) = \sum_{s_n^i} min\{c_s^i, c_q^i\}$.

With respect to the edit distance, the result of the match-count model satisfies the following theorem.

Theorem 5.1: If the edit distance between S and Q is τ, then the return of the match-count model has $MC(G(S), G(Q)) \geq \max\{|Q|, |S|\} - n + 1 - \tau * n$. [20]

According to Theorem 5.1, we can use the result of the match-count model as an indicator for selecting candidates for the query. Our strategy is to retrieve $\mathbf{K}$ candidates from GENIE according to match count with a large $\mathbf{K}(\mathbf{K} >> k)$. Then we can employ a verification process to calculate the edit distance between the query Q and the $\mathbf{K}$ candidates to obtain the k-th most similar sequence $S^{\mathbf{k}'}$. The detail of the verification process is shown in Algorithm 2 in [10].

With this method, we can know whether the real top-k sequences are correctly returned by GENIE, though we cannot guarantee the returned top-k candidates are the real top-k data sequence for all queries. In other words, after the verification, we can know whether $S^{\mathbf{k}'}$ is the real k-th most similar sequence of Q accroding to the following theorem.

Theorem 5.2: For the $\mathbf{K}$-th candidates $S^{\mathbf{K}}$ returned by GENIE according to count, suppose the match count between $S^{\mathbf{K}}$ and query Q is $c_{\mathbf{K}} = MC(G(S^{\mathbf{K}}), G(Q))$. Among the $\mathbf{K}$ candidates, after employing the verification algorithm, we can obtain the edit distance between k-th most similar sequence (among the $\mathbf{K}$ candidates) and Q is $\tau_{\mathbf{k}'} = ed(Q, S^{\mathbf{k}'})$. If $c_{\mathbf{K}} < |Q| - n + 1 - \tau_{\mathbf{k}'} * n$, then the real top-$k$ results are correctly returned by GENIE.

A possible solution for sequence similarity search is to repeat the search process by GENIE with larger $\mathbf{K}$, until the condition in Lemma 5.2 is satisfied. In the worst case it may need to scan the whole data set before retrieving the real top-k sequences under edit distance. However, as shown in our experiment, it can work well in practice for near edit distance similarity search in some applications.

B. Searching on short document data

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

We can explain the result returned by GENIE on short document data by the document vector space model. Documents can be represented by a binary vector space model where each word represents a separate dimension in the vector. If a word occurs in the document, its value in the vector is one, otherwise it is zero. The output of the match-count model, which is the number of word co-occurring 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.

C. Searching on relational data

GENIE can also be used to support queries on relational data. In Fig. 1, we have shown how to build an inverted index for relational tuples. For attributes with continuous value, we assume that they can be discretized to an acceptable granularity level. A range selection query on a relational table is a set of specific ranges on attributes of the relational table.

The top-k result returned by GENIE on relational tables can be considered a special case of the traditional top-k selection query. The top-k selection query selects the k tuples in a relational table with the largest predefined ranking score function $F(\cdot)$ (SQL *ORDER BY* $F(\cdot)$). 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.

VI. Experiments

A. Settings

1) Datasets: We use five real-life datasets to evaluate our system. Each dataset corresponds to one similarity measure respectively introduced in Section IV and Section V.

[**OCR**][3] This is a dataset for optical character recognition. It contains 3.5M data points and each point has 1156 dimensions. The size of this dataset is 3.94 GB. We randomly select 10K points from the dataset as query/test set (and remove them from the dataset). We use RBH to generate the LSH signature, which is further re-hashed into an integer domain of [0,8192).

[**SIFT**][4] This dataset [21] contains 4.5M SIFT features which are 128-dimensional points. Its total size is 1.49 GB. We randomly select 10K features as query set and remove them from the dataset. We select the hash functions from E2LSH family [6] and each function transforms a feature into 67 buckets. The setting of bucket width follows the routine in [6]. We use this dataset to evaluate the ANN search in high dimensional space.

[**SIFT_LARGE**] [5] To evaluate the scalability of our system, we also extract 36 millions SIFT features by ourselves from the ILSVRC-2010 image dataset. The size of this dataset is 14.75 GB. We use the same method as described above for SIFT to process the data.

[**DBLP**][6] This dataset is obtained by extracting article titles from the DBLP website. The total number of sequences is 5.0M and the size of this dataset is 0.94 GB. We randomly choose 10K sequences as a test data, and then modify 20% of the characters of the sequences. This dataset is to serve the experiment of sequence similarity search in Section V-A. Specially, we set $\mathbf{K} = 32$ and $k = 1$ by default.

[**Tweets**][7] This dataset has 6.8M tweets. We remove stop words from the tweets. The dataset is crawled by our collaborators from Twitter for three months by keeping the tweets containing a set of keywords.[8] The data size is 0.46 GB. We reserve 10K tweets as a query set. It is used to study the short document similarity search (see Section V-B).

[**Adult**][9] This dataset has census information [22] which contains 49K rows with 14 attributes (mixed of numerical and categorical ones). For numerical data, we discretize all value into 1024 intervals of equal width. We further duplicate every row 20 times. Thus, there are 0.98M instances (with size being 5.8 GB). We select 10K tuples as queries. For numerical attributes, the query item range is defined as $[discretized\_value - 50, discretized\_value + 50]$. We use it to study the selection from relational data (see Section V-C).

2) Competitors: We use the following competitors as baselines to evaluate the performance of GENIE.

[**GPU-LSH**] We use GPU-LSH [4], [23] as a competitor of GENIE for ANN search in high dimensional space and its source code is publicly available[10]. Furthermore, since there is no GPU-based LSH method for ANN search in Laplacian kernel space, we still use GPU-LSH method as a competitor for ANN search of GENIE. We configure the parameters of GPU-LSH to make sure its ANN search results have similar quality as GENIE, which is discussed in Section VI-D1. We only use 1M data points for GPU-LSH on OCR dataset since GPU-LSH cannot afford more data points.

[**GPU-SPQ**] We implemented a priority queue-like method on GPU as a competitor. We first scan the whole dataset to compute match-count values between queries and all points, and store these computed results in an array. Then we use a GPU-based fast k-selection [9] method to extract the top-k candidates from the array for each query. We name this top-k calculation method as SPQ (which denotes GPU fast

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

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

[5] http://image-net.org/challenges/LSVRC/2010/index#data

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

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

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

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

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

k-selection from an array as a priority queue). We give an introduction to SPQ in [10]. Note that for ANN search, we scan on the LSH signatures (not original data).

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

[**CPU-LSH**] We use CPU-LSH [24] for ANN search in high dimensional space as a competitor after obtaining its source code from authors' website[11]. We use the suggestion method in the paper to determine the parameters.

[**AppGram**] This is one of the state-of-the-art methods for sequence similarity search under edit distance on the CPU [2]. We use AppGram as a baseline for comparing the running time of GENIE for sequence similarity search. Note that AppGram and GENIE are not completely comparable, since AppGram tries its best to find the true kNNs, while GENIE only does one round search process in the experiment. Thus some true kNNs may be missed (though we know which queries do not have true top-k, and another search process can be issued to explore the true kNNs). We give more discussion about this in Section VI-D2.

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

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

Unless otherwise specified, we set $k = 100$ and set the submitted query number per batch to the GPU as 1024. All the reported results are the average of running results of ten times. By default, we do not enable the load balancing function since it is not necessary when the query number is large for one batch process (the experiment about load balancing can be found in [10]). For ANN search, we use the method introduced in Section IV-B1 to determine the number of LSH hash functions with setting $\epsilon = \delta = 0.06$, therefore the number of hash functions is $m = 237$.

B. Efficiency of GENIE

1) Search time for multiple queries: We compare the running time among GENIE and its competitors. We do not include index building time for all competitors since index building can be done offline. The index building time of GENIE is discussed in Section VI-B2.

We show the total running time with respect to different numbers of queries in Fig. 8 (y-axis is log-scaled). Our method outperforms GPU-SPQ by more than one order of magnitude, and it can achieve more than two orders of magnitude over GPU-SPQ and AppGram for sequence search. Furthermore, GPU-SPQ can only run less than 256 queries in parallel (except for Adult dataset) for one batch process, but GENIE can support more than 1000 queries in parallel.

[11] http://ss.sysu.edu.cn/~fjl/c2lsh/C2LSH_Source_Code.tar.gz

TABLE I
TIME PROFILING OF DIFFERENT STAGES OF GENIE FOR 1024 QUERIES (THE UNIT OF TIME IS $second$).

Stage		OCR	SIFT	DBLP	Tweets	Adult
Index build		81.39	47.73	147.34	12.10	1.06
Index loading		0.53	0.34	0.20	0.088	0.011
Query	transfer	0.015	0.018	0.0004	0.0004	0.0004
	match	2.60	7.04	0.85	1.19	1.82
	selection	0.004	0.003	0.11*	0.003	0.009

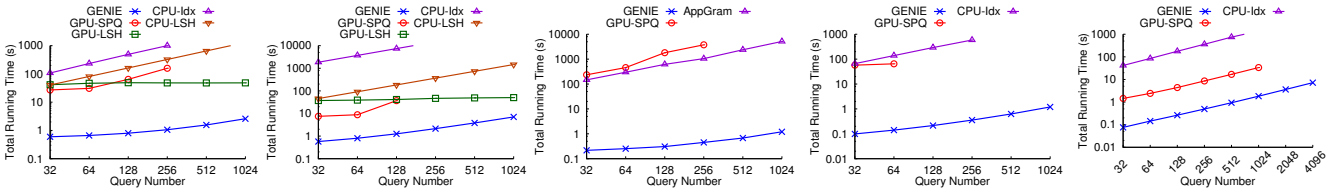
*This includes verification time which is the major cost.

As we can see from Fig. 8, GENIE can also outperform GPU-LSH about one order of magnitude. The running time of GPU-LSH is relatively stable with varying numbers of queries. This is because GPU-LSH uses one thread to process one query, thus, GPU-LSH achieves its best performance when there are 1024 queries (which is the maximum number of threads per block on the GPU). Note that we only use 1M data points for GPU-LSH on OCR dataset.

Fig. 9 conveys the running time of GENIE and its competitors with varying numbers of data points for each dataset. Since most of the competitors cannot run 1024 queries for one batch, we fix the query number as 512 in this experiment. The running time of GENIE is gradually increased with the growth of data size. Nevertheless, the running time of GPU-LSH is relatively stable on all datasets with respect to the data size. The possible reason is that GPU-LSH uses many LSH hash tables and LSH hash functions to break the data points into short blocks, therefore, the time for accessing the LSH index on the GPU becomes the main cost of query processing.

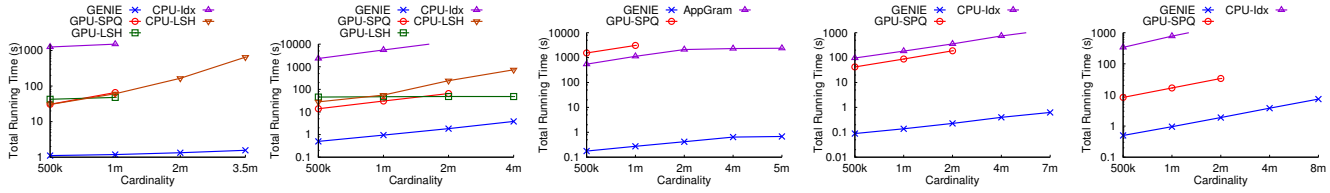
Fig. 11 shows the running time of GENIE and GPU-LSH for a larger number (up to 65536) of queries on SIFT data. Though GPU-LSH can support ten thousands of queries per batch, GENIE can also support the such large number of queries with breaking query set into several small batches. With setting 1024 queries as a batch for GENIE, we can see that the time cost of GPU-LSH to process 65536 queries with one batch is 1329 seconds, while GENIE can process the same number of queries (with 64 batches) in 441 seconds.

2) Time profiling: Table I shows the time cost for different stages of GENIE. The "Index-build" represents the running time to build the inverted index on the CPU. This is an one-time cost, and we do not count it in the query time. The "Index-loading" displays the time cost to swap the inverted index from the CPU to the GPU. The rows of "Query" display the time for similarity search with 1024 queries per batch. The "Query-transfer" is the time cost to transfer queries and other information from the CPU to the GPU. The "Query-selection" contains the time for selecting candidates from c-PQ and sending back the candidates to the CPU (For DBLP data, it also includes the time of verification). The "Query-match" is the time cost for scanning inverted index which dominates the cost for similarity search. This confirms our design choice of using GPU to accelerate this task.



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

Fig. 8. Total running time for multiple queries.



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

Fig. 9. Varying data size for multiple queries (The query number is 512).

TABLE II
RUNNING TIME OF GENIE WITH MULTIPLE LOADINGS ON SIFT_LARGE DATASET FOR 1024 QUERIES(UNIT: SECOND)

SIFT_LARGE	6M	12M	24M	36M
GENIE	4.32	8.62	17.26	25.90
GPU-LSH	47.71	48.82	(97.64)*	(146.46)*
CPU-LSH	2817	5644	12333	20197

*It is estimated with multiple loading method.

TABLE III
EXTRA RUNNING TIME COST OF GENIE WITH MULTIPLE LOADINGS WITH THE SAME SETTING OF TABLE II(UNIT: SECOND)

SIFT_LARGE	6M	12M	24M	36M
Index loading	0.50	1.03	2.01	3.02
Result merging	0	0.04	0.10	0.22
GENIE_total	4.32	8.62	17.26	25.90

3) Searching on large data with multiple loadings: If the data set is too large to be processed with limited GPU memory, we adopt a multiple loading method (see Section III-D). Table II shows the scalability of GENIE with different data sizes on SIFT_LARGE dataset. In this experiment, we set the data part for each loading as 6M data points. By resorting to multiple loadings, GENIE can finish the query process for 1024 queries with 25.90 seconds on 36M SIFT data points. It also shows that GENIE with multiple loadings can scale up linearly with the number of data points. Since GPU-LSH cannot handle datasets with larger than 12M points, we estimate the running time of GPU-LSH on 24M and 36M data points with the same multiple loading method but without including index loading and result merge time. We can see that GPU-LSH has almost six times of running time of GENIE for the same dataset.

GENIE with multiple loadings has two extra steps: 1) index loading: swapping index of each data part into the GPU memory and 2) result merging: merging the query results of each data part to obtain the final result. The running time cost of each extra step is shown in Table III. We can see that the extra steps only take a small portion of the total time cost.

4) Discussion: Here we give a brief discussion about the root causes that GENIE outperforms other methods. It is not surprising that GENIE can outperform all the CPU-based algorithms like CPU-LSH, CPI-Idx and AppGram significantly.

The key reason that GENIE can outperform GPU-LSH is due to the novel structure of c-PQ for candidate selection. The main bottleneck of GPU-LSH is to select top-k candidates from the candidate set generated by LSH, whose method essentially is to sort all candidates which is an expensive computation. Meanwhile, c-PQ can obtain the candidates by scanning the small Hash Table once.

TABLE IV
MEMORY CONSUMPTION PER QUERY (UNIT: MB)

dataset	OCR	SIFT	DBLP	Tweets	Adult
GENIE	4.9	6.5	7.2	10.2	1.4
GEN-SPQ	41.0	49.1	47.3	61.4	8.8

GENIE outperforms GPU-SPQ with similar reasons. GPU-SPQ uses a k-selection algorithm on the GPU which requires multiple iterations to scan all candidates. Whereas GENIE only needs to scan the Hash Table once whose size (which is $O(k*AT)$) is much smaller than the candidate set of GPU-SPQ.

C. Effectiveness of c-PQ

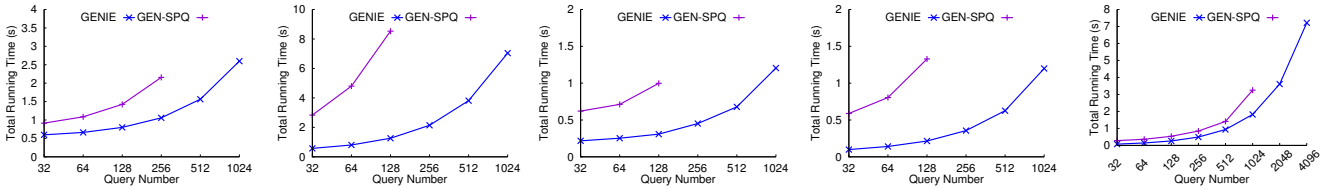
c-PQ can significantly reduce the memory requirement and the running time cost for GENIE. In Fig. 10, GEN-SPQ represents the running time of GENIE without c-PQ. We can see that, when the number of queries is the same, with the help of c-PQ the running time of GENIE decreases significantly since it avoids selecting candidates from a large Count Table.

From Table IV we can see that GENIE reduces memory consumption per query to $1/5 \sim 1/10$ of the one of GEN-SPQ. To evaluate the memory consumption, with fixing the data size, we gradually increase the number of queries to find the maximum query number handled by our GPU, then we calculate the memory consumption per query by using 12 GB to divide the maximum query number.

D. Effectiveness of GENIE

In this section, we evaluate the effectiveness of GENIE under the LSH scheme and the SA scheme.

1) ANN Search with GENIE: Here we discuss the quality of the ANN search with GENIE and GPU-LSH. The used evaluation metric is *approximation ratio*, which is defined as how many times farther a reported neighbor is compared to the real nearest neighbor. When to evaluate the running time, we configure the parameters of GPU-LSH and GENIE to ensure that they have similar approximation ratio. A detailed



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

Fig. 10. The effectiveness of c-PQ.

TABLE V
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

TABLE VI
ACCURACY OF TOP-1 SEARCH ON DBLP DATASET ON GENIE (QUERY LENGTH=40 AND $\mathbf{K} = 32$)

Percent of modified	0.1	0.2	0.3	0.4
Accuracy	1.0	0.999	0.995	0.954
Latency tiem (s)	1.3	1.2	1.1	1.2

description of approximation ratio and the parameter setting method can be found in [10].

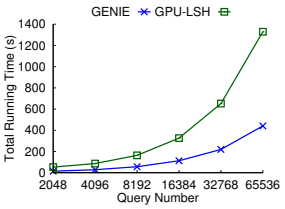


Fig. 11. Running time with a large number of queries on SIFT data

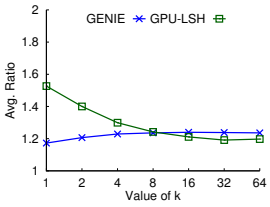


Fig. 12. Approximation ratio v.s. value of k on SIFT data

Fig. 12 shows the approximation ratio of GPU-LSH and GENIE, where GENIE has stable approximation ratio with varying k; whereas GPU-LSH has large approximation ratio when k is small. The increase of approximation ratio of GPU-LSH with smaller k is a common phenomenon which also appears in some pervious LSH methods like [25]. The reason is that these methods usually adopt some early-stop conditions, thus with larger k they can access more points to improve the approximation ratio [25].

We use a similar method to determine the parameters for GPU-LSH and GENIE on the OCR dataset. GPU-LSH uses GPU's constant memory to store random vectors for LSH. Thus, the number of hash functions on OCR data cannot be larger than 8 otherwise the constant memory overflows. We use only 1M data points from the OCR dataset for GPU-LSH since it cannot work on a 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 V where the number of hash tables for GPU-LSH is set as 100. Note that the prediction performance of GPU-LSH is slightly worse than the one of GENIE. It is possible to improve the performance of GPU-LSH by increasing the number of hash tables, which will dramatically increase the running time for queries.

2) Sequence similarity search with GENIE: After finishing the search on GENIE, we can identify that some queries do not obtain real top-k results (see Section V-A). Table VI shows the percent of the queries obtaining correct top-1 search results with one round of the search process for 1024 queries. As we see from Table VI, with less than 10% modification, GENIE can return correct results for almost all queries. Even with 40% modification, GENIE can still return correct results for more than 95% of queries. A typical application of such sequence similarity search is (typing) sequence error correction, where GENIE can return the most similar words (within minimum edit distance in a database) for the 1024 queries with a latency time of 1 second (as shown in Fig. 8 and Table I). Note that the one second is the whole latency time for 1024 queries including index loading, query transfer, matching in GENIE and verification. AppGram may do this job with better accuracy, but it has much larger latency. A discussion on multiple round search and how to set $\mathbf{K}$ is in [10].

VII. RELATED WORK

A. Similarity search on different data

Due to the "curse of dimensionality", spatial index methods 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 LSH scheme for ANN search in high dimensional space [16]. We refer interested readers to a survey of LSH [17].

The similarity between sets, feature sketches and geometries is often known only implicitly, thus the computable kernel function is adopted for similarity search. To scale up the similarity search, the LSH-based ANN search in such kernel spaces has drawn considerable attention. Charikar [5] investigates several LSH families for kernelized similarity search. Wang et al. [17] give a good survey about the LSH scheme on different data types. GENIE can support the similarity search in an arbitrary kernel space if it has an LSH scheme.

There is a wealth of literature concerning similarity search on complex structured data, and a large number of indexes have been devised. Many of them adopt the SA scheme [7], [8]. Different data types are broken down into different types of sub-units. Examples include words for documents, n-grams for sequences [2], binary branches for trees [12] and stars for graphs [11]. Sometimes, a verification step is necessary to compute the real distance (e.g. edit distance) between the candidates and the query object [2], [11], [12].

B. Parallelizing similarity search

Parallelism can be adopted to improve the throughput for similarity search. There are also some proposed index structures on graphs and trees that can be parallelized [26], [11]. However, indexes tailored to special data types cannot be easily extended to support other data types. There are a few GPU-based methods for ANN search using LSH. Pan et al. [4],

[23] propose a searching method on the GPU using a bi-level LSH algorithm, which specially designed for ANN search in the $l_\mathfrak{p}$ space. However GENIE can generally support LSH for ANN search under various similarity measures.

C. Data structures and models on the GPU

There are some works [3], [27] about inverted index on the GPU to design specialized algorithms for accelerating some important operations on search engines. An inverted-like index on the GPU is also studied for continuous time series search [28]. As far as we known, there is no existing work on inverted index framework for generic similarity search on the GPU.

Tree-based data structures are also investigated to utilize the parallel capability of the GPU [1]. Some GPU systems for key-value store are also studied [29], [30].

D. Frequent item finding algorithm

Some previous work related to Count Priority Queue (c-PQ) of GENIE is frequent item finding algorithm [31], which can be categorized as counter-based approach (like LossyCounting [32] and SpaceSaving [33]) and sketch-based approach (like Count-Min [34] and Count-Sketch [35]). However, both approaches are approximation methods, whereas c-PQ can return the exact top-k frequent items. Moreover, several frequent item finding algorithms (like SpaceSaving and Count-Min) require priority queue-like operations, making them nontrivial be implemented on the GPU.

VIII. Conclusion

In this paper, we presented GENIE, a generic inverted index framework, which tries to reduce the programmer burden by providing a generic fashion for similarity search on the GPU for data types and similarity measures that can be modeled in the match-count model. Several techniques are devised to improve the parallelism and scaling out of the GPU, like c-PQ to reduce the time cost and the multiple loading method for handling large datasets. We also proved that GENIE can support τ-ANN search for any similarity measure satisfying the LSH scheme, as well as similarity search on original data with the SA scheme. In particular, we investigated how to use GENIE to support ANN search in kernel space and in high dimensional space, similarity search on sequence data and document data, and top-k selection on relational data. Extensive experiments on various datasets demonstrate the efficiency and effectiveness of GENIE.

Acknowledgments

This research was carried out at the SeSaMe Centre. It is supported by the Singapore NRF under its IRC@SG Funding Initiative and administered by the IDMPO. The work by H. V. Jagadish was partially supported by the US National Science Foundation under Grants IIS-1250880 and IIS-1741022.

References

[1] L. Luo, M. D. Wong, and L. Leong, "Parallel implementation of r-trees on the gpu," in *ASP-DAC*, 2012, pp. 353–358.
[2] X. Wang, X. Ding, A. K. Tung, and Z. Zhang, "Efficient and effective knn sequence search with approximate n-grams," *PVLDB*, vol. 7, no. 1, pp. 1–12, 2013.
[3] S. Ding, J. He, H. Yan, and T. Suel, "Using graphics processors for high performance ir query processing," in *WWW*, 2009, pp. 421–430.
[4] J. Pan and D. Manocha, "Fast gpu-based locality sensitive hashing for k-nearest neighbor computation," in *GIS*, 2011, pp. 211–220.
[5] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *STOC*, 2002, pp. 380–388.
[6] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *SoCG*, 2004.
[7] S. Aparicio, J. Chapman, E. Stupka, N. Putnam, J.-m. Chia *et al.*, "Whole-genome shotgun assembly and analysis of the genome of fugu rubripes," *Science*, vol. 297, no. 5585, pp. 1301–1310, 2002.
[8] 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*, vol. 431, no. 7011, pp. 927–930, 2004.
[9] T. Alabi, J. D. Blanchard, B. Gordon, and R. Steinbach, "Fast k-selection algorithms for graphics processing units," *JEA*, vol. 17, pp. 4–2, 2012.
[10] J. Zhou, Q. Guo, H. V. Jagadish, L. Krčál, S. Liu, W. Luan, A. Tung, Y. Yang, and Y. Zheng, "A generic inverted index framework for similarity search on the gpu – techincal report," *arXiv:1603.08390*, 2018.
[11] X. Yan, P. S. Yu, and J. Han, "Substructure similarity search in graph databases," in *SIGMOD*, 2005, pp. 766–777.
[12] R. Yang, P. Kalnis, and A. K. Tung, "Similarity evaluation on tree-structured data," in *SIGMOD*, 2005, pp. 754–765.
[13] X. He, D. Agarwal, and S. K. Prasad, "Design and implementation of a parallel priority queue on many-core architectures," in *HiPC*, 2012.
[14] I. García, S. Lefebvre, S. Hornus, and A. Lasram, "Coherent parallel hashing," *ACM TOG*, vol. 30, no. 6, p. 161, 2011.
[15] M. Moazeni and M. Sarrafzadeh, "Lock-free hash table on graphics processors," in *SAAHPC*, 2012, pp. 133–136.
[16] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *STOC*, 1998, pp. 604–613.
[17] J. Wang, H. T. Shen, J. Song, and J. Ji, "Hashing for similarity search: A survey," *arXiv:1408.2927*, 2014.
[18] A. Rahimi and B. Recht, "Random features for large-scale kernel machines," in *NIPS*, 2007, pp. 1177–1184.
[19] V. Satuluri and S. Parthasarathy, "Bayesian locality sensitive hashing for fast similarity search," *PVLDB*, vol. 5, no. 5, pp. 430–441, 2012.
[20] E. Sutinen and J. Tarhio, "Filtration with q-samples in approximate string matching," in *Combinatorial Pattern Matching*, 1996, pp. 50–63.
[21] H. Jegou, M. Douze, and C. Schmid, "Hamming embedding and weak geometric consistency for large scale image search," in *ECCV*, 2008.
[22] M. Lichman. (2016) UCI machine learning repository. http://archive.ics.uci.edu/ml.
[23] J. Pan and D. Manocha, "Bi-level locality sensitive hashing for k-nearest neighbor computation," in *ICDE*, 2012, pp. 378–389.
[24] J. Gan, J. Feng, Q. Fang, and W. Ng, "Locality-sensitive hashing scheme based on dynamic collision counting," in *SIMOD*, 2012, pp. 541–552.
[25] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin, "Srs: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index," *PVLDB*, vol. 8, no. 1, pp. 1–12, 2014.
[26] S. Tatikonda and S. Parthasarathy, "Hashing tree-structured data: Methods and applications," in *ICDE*, 2010, pp. 429–440.
[27] 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*, pp. 470–481, 2011.
[28] J. Zhou and A. K. Tung, "Smiler: A semi-lazy time series prediction system for sensors," in *SIGMOD*, 2015, pp. 1871–1886.
[29] 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*, vol. 8, no. 11, pp. 1226–1237, 2015.
[30] 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*, 2012, pp. 88–98.
[31] G. Cormode and M. Hadjieleftheriou, "Finding the frequent items in streams of data," *CACM*, vol. 52, no. 10, pp. 97–105, 2009.
[32] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *VLDB*, 2002, pp. 346–357.
[33] A. Metwally, D. Agrawal, and A. E. Abbadi, "An integrated efficient solution for computing frequent and top-k elements in data streams," *TODS*, vol. 31, no. 3, pp. 1095–1133, 2006.
[34] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.
[35] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *TCS*, vol. 312, no. 1, pp. 3–15, 2004.