open proceedings

Efficient Processing of Hamming-Distance-Based Similarity-Search Queries Over MapReduce *

Mingjie Tang †, Yongyang Yu †, Walid G. Aref †, Qutaibah M. Malluhi ‡, Mourad Ouzzani ⋆

† *Computer Science, Purdue University,* ‡ *Qatar University,* ⋆ *Qatar Computing Research Institute*

{tang49,yu163,aref}@cs.purdue.edu, qmalluhi@qu.edu.qa, mouzzani@qf.org.qa

ABSTRACT

Similarity search is crucial to many applications. Of particular interest are two flavors of the Hamming distance range query, namely, the Hamming select and the Hamming join (Hamming-select and Hamming-join, respectively). Hamming distance is widely used in approximate near neighbor search for high dimensional data, such as images and document collections. For example, using predefined similarity hash functions, high-dimensional data is mapped into one-dimensional binary codes that are, then linearly scanned to perform Hamming-distance comparisons. These distance comparisons on the binary codes are usually costly and, often involves excessive redundancies. This paper introduces a new index, termed the *HA-Index*, that speeds up distance comparisons and eliminates redundancies when performing the two flavors of Hamming distance range queries. An efficient search algorithm based on the *HA-index* is presented. A distributed version of the *HA-index* is introduced and algorithms for realizing Hamming distance-select and Hamming distance-join operations on a MapReduce platform are prototyped. Extensive experiments using real datasets demonstrates that the *HA-index* and the corresponding search algorithms achieve up to two orders of magnitude speedup over existing state-of-the-art approaches, while saving more than ten times in memory space.

1. INTRODUCTION

Hamming-distance search over big data plays an important role in a large variety of applications. For example, widely used search engines, such as Google, Baidu, and Bing, use Hamming-distance search in their image content-based search engines that usually index billions of images (e.g., refer to [1]). Typically, each image is modeled by a high-dimensional vector of extracted features, e.g., color histograms, texture features, and edge orientation. Then, based on the learned similarity hash function, e.g., as in [1, 2, 3], each image is converted into a binary code. Given a query image that gets modeled with the same high-dimensional vector of features, the search engine maps it into a binary code and performs a Hamming-distance search to find images whose binary codes have a Hamming distance smaller than a given threshold $\hat{h}$ from the query image. Hamming search is also widely used to detect duplicate web pages in applications, e.g., web mirroring, plagiarism, and spam detection [4]. A similarity hash function [5] is applied to map a high-dimensional vector into a binary code, then a Hamming-distance range search finds web documents that are similar to the query document.

Typically, computing the Hamming distance between two binary codes is performed by an Exclusive-Or operation (XOR, for short) that is followed by a count operation to sum up the number of ones in the XOR result. The number of ones corresponds to the number of differing bits between the two binary codes. Thus, a linear scan over the binary codes of the underlying dataset takes place to perform the XORing, the counting, and the ranking to retrieve the objects within a certain range of t_q (i.e., the ones within the predefined Hamming distance threshold $\hat{h}$). Due to the linear scan, this approach is slow. When joining two tables via a Hamming distance predicate, the linear scan approach induces a quadratic cost to evaluate the join. An efficient indexing of the binary codes is called for to perform the Hamming range query and avoid a complete scan over the underlying dataset, while remaining low on memory usage.

The Hamming distance problem [6, 7] is first studied for small distance thresholds, i.e., $\hat{h} = 1$. An algorithm proposed by Manku et al. [4] uses multiple hash tables to enhance query speed. However, duplicating the hash entries multiple times for the entire datasets is expensive and performance tends to degrade as a linear scan over tuples within a bucket is required. HEngine [8] extends Manku's algorithm to improve the query's speed with less memory. However, HEngine is sensitive to the Hamming distance threshold $\hat{h}$, and it needs to generate one-bit differing binary code with each query, then carry out several binary searches over sorted hash tables. Recently, MapReduce as a reliable distributed computing model has been adopted for handling a variety of similarity queries, e.g., [4, 9, 10, 11, 12]. Existing techniques for Hamming-distance queries cannot be easily extended for MapReduce. The reason is that most of the existing techniques use centralized multiple hash-table indexes. Because MapReduce needs to write intermediate data on disk when shuffling data between the mappers and the reducers, rearranging multiple indexes and multiple versions of the same data can be quite inefficient.

In this paper, we focus on two variants of the Hamming distance query problem, namely Hamming-distance-based select and Hamming-distance-based join (for short, Hamming-select and Hamming-join, respectively). We propose a new index, termed the HA-Index, that is designed to reduce redundant and duplicate dis-

*This work was supported by an NPRP grant 4-1534-1-247 from the Qatar National Research Fund and by the National Science Foundation Grants IIS 0916614, IIS 1117766, and IIS 0964639.

(c) 2015, Copyright is with the authors. Published in Proc. 18th International Conference on Extending Database Technology (EDBT), March 23-27, 2015, Brussels, Belgium: ISBN 978-3-89318-067-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0
.

10.5441/002/edbt.2015.32

tance computations during the Hamming-distance search. The HA-Index assumes that the underlying datasets are preprocessed; data is mapped from the high-dimensional space into one-dimensional binary codes that are fixed-length strings of 0's and 1's. Then, the binary codes are sorted using the Gray ordering [13]. Sorting the binary codes in this way helps group together multiple binary codes that share a common substring or non-contiguous yet similar sequences of bits. By computing the distances between the query binary code and similar substrings, many redundant distance computations can be avoided.

The contributions of this paper are as follows.

- Based on properties of binary codes, we introduce two approaches to improve the performance of Hamming-select and Hamming-join. The first approach uses a simple Radix-tree index from the literature. The second approach is based on the HA-Index with both a static and a dynamic version. We also introduce the maintenance operations, i.e., build, insert, update, and search operations, for the dynamic HA-Index.
- For Hamming-joins over large and skewed data, we propose an efficient data partitioning technique for balancing data computations among servers, and introduce a distributed version of the HA-Index to reduce data shuffling inside MapReduce.
- We conduct an extensive experimental study using real datasets and demonstrate that the HA-Index (i) enhances the performance of Hamming-select and Hamming-join by two orders of magnitude over state-of-the-art techniques, and (ii) saves memory usage by more than one order of magnitude. We also evaluate how the proposed index improves approximate algorithms for kNN-select and kNN-join operations.

The rest of this paper proceeds as follows. Section 2 discusses related work. Section 3 presents the problem definition. Section 4 introduces the centralized-server approach for approximate Hamming-select and Hamming-join. Section 5 introduces the distributed version of the HA-Index using MapReduce and explains how Hamming-select and Hamming-join can be performed in MapReduce. Section 6 presents and discusses the experimental results. Finally, Section 7 contains concluding remarks.

Table 1: Symbols and their definitions

Symbol	**Definition**
$\mathbb{R}^d$	d-dimensional vector space
n, $\lvert R\rvert$	Number of tuples in dataset R
m, $\lvert S\rvert$	Number of tuples in dataset S
t_q	Query tuple
k	The required number of nearest neighbors
$\lvert\lvert t_i, t_j\rvert\rvert_h$	Hamming distance between tuples t_i and t_j
H	Similarity Hash function
U_i	Binary code for tuple t_i
$L = \lvert U\rvert$	Length of the binary code U
l_i	The ith bit in the binary code
$\hat{h}$	Hamming distance threshold
$\hat{h}$-select(t_q, S)	Hamming distance select for tuple t_q and datasets S
$\hat{h}$-join(R, S)	Hamming distance join between datasets R and S
N	Number of data partitions

2. RELATED WORK

Using the Hamming distance as a similarity metric has been studied in the theory community, e.g., [6, 7]. When the Hamming-distance query threshold is small, i.e., $\hat{h} = 1$, Yao et.al [7] propose an algorithm with $O(m \log\log(n))$ query time and $O(nm \log(m))$ space. Yao's algorithm recursively cuts the query binary code and each binary code in the dataset in half, and then finds exact matches in the dataset for the left or the right half of the query binary code. [14] demonstrates that similarity search in chemical information via the Tanimoto Similarity metric can be transformed into a Hamming-distance query.

Hamming-distance queries are attracting more attention for processing large volumes of data. A relatively recent work [4] uses multiple hash tables, and hence more space, to reduce the linear computation of the Hamming distance during query time. The idea behind this approach is that if two binary codes are within a Hamming distance $\hat{h}$, then at least one of the $\hat{h}$+1 segments are exact matches for two binary codes. This algorithm needs to replicate the database multiple times, and it sorts each copy based on parts of the segments. The Hamming-distance computation is still performed in a linear fashion over tuples of the same bucket in a certain hash table. Thus, it fails to scale as data size increases. For performing a Hamming-join of two datasets, say R and S, [4] extends the sequential approach to MapReduce by broadcasting Table R into each server, then applying a sequential algorithm between R and S. This approach is subject to a very heavy shuffling cost and servers cannot work in a load-balanced way when data is skewed. HEngine [8] adopts a similar idea to that in [4], but uses approximate matching instead by generating multiple one-bit difference binary codes. The HEngine uses less memory but achieves limited performance speedup. HmSearch [14] is an exact matching approach that index over signature of the binary codes. The size of the index increases dramatically, because HmSearch need to generated large amount of unique signatures. If used in the context of MapReduce, the shuffling cost between the mappers and the reducers is expected to be expensive. Our proposed HA-Index extracts and groups similar binary codes from among the various tuples to reduce the cost of shuffling and hence is applicable to MapReduce as we illustrate later in this paper. Through data sampling, we partition that data in a way that uniformly distributes the dataset among the reducer servers and hence enables better load balancing. Experimental comparison with [4, 8] shows that our proposed HA-Index is two orders of magnitude faster and uses ten times less memory as illustrated in the experimental section of this paper.

Two related and popular operations to Hamming distance queries are the k-nearest-neighbor select (kNN-select) and k-nearest-neighbor join (kNN-join) [15, 16]. Given a dataset, say S, and a query focal point, say t_q, kNN-select finds in S the k-nearest-neighbors to t_q. Given two datasets, say R and S, R kNN-join S finds the k-nearest-neighbors in S for each tuple in R. In high-dimensional spaces and because of the curse of dimensionality [17], data-independent hash-based approximate kNN (e.g., locality sensitive hashing (LSH) [18]) has attracted attention as it can speed-up query execution while having acceptable error margins. Recently, data-dependent hashing has been proposed to learn the hash function, say $H()$, given the underlying dataset, e.g., as in [2]. There has been a plethora of work in learning good and representative hash functions, e.g., [2, 3, 1]. Given the learned similarity hash function $H()$, a tuple, say t_i, is mapped into its binary code,

say U_i, i.e., $H(t_i) = U_i$. Afterwards, all the binary codes of the dataset R are scanned to find data tuples that are different from the query's binary code U_i by at most $\hat{h}$ bit-positions. If the answer set size is more than k, then only the k-closest answers are retained. However, if the size of the result set is less than k, then a larger distance threshold is estimated and the near neighbor query is repeated. The process is stopped when k or more answers are reported. Notice that the core of the method for approximate kNN search is a Hamming-distance query with a threshold $\hat{h}$. In our experiments, we use the state-of-the-art approach [2] to learn the hash function, and show how our proposed approach can speed up approximate kNN-select and kNN-join.

3. PRELIMINARIES

3.1 Hamming-distance-based Similarity Operations

We assume that data tuples represent points in a d-dimensional metric space, say $\mathbb{R}^d$. Given two data tuples, say t_i and t_j, let $||t_i, t_j||$ be the distance between t_i and t_j in $\mathbb{R}^d$. The Hamming distance between t_i and t_j, denoted by $||t_i, t_j||_h$, helps in retrieving the tuples in a dataset that are within some threshold from an input tuple, either t_i or t_j in this case. Table 1 summarizes the symbols used in this paper.

DEFINITION 1. *Hamming-distance-based Similarity Select [4] (referred to as Hamming-select, for short): Given a query tuple, say t_q, and a dataset, say S, with its corresponding collection of binary codes, denoted by U_S, and an integer, say $\hat{h}$, that represents the similarity threshold for the Hamming distance, Hamming-select identifies a subset from S, denoted by $\hat{h}$-select(t_q, S) for short, where $\forall o \in \hat{h}$-select$(t_q, S), ||o, t_q||_h \leq \hat{h}$.*

Similarly, we define the Hamming-distance-based similarity join as follows.

DEFINITION 2. *Hamming-distance-based Similarity Join (referred to as Hamming-join, for short): Given two collections of binary codes, say U_R and U_S, that correspond to two datasets, say R and S, respectively, and an integer, say $\hat{h}$, that represents the similarity threshold for the Hamming distance, Hamming-join identifies the set $\hat{h}$-join(R, S) of tuple pairs such that $(t_i, t_j) \in \hat{h}$-join(R, S) iff $t_i \in R$ and $t_j \in S$ and $||t_i, t_j||_h \leq \hat{h}$.*[1]

EXAMPLE 1. *Consider the set of binary codes given in Table 2a and a Hamming distance threshold $\hat{h} = 3$. The query tuple t_q has a binary code "101100010". The output of the Hamming-distance-based similarity select is $\{t_0, t_3, t_4, t_6\}$. Using the same Hamming distance threshold $\hat{h}$, the output of the Hamming-distance-based similarity join for the datasets in Tables 2b and 2a is $\{(r_0, t_0), (r_0, t_3), (r_0, t_4), (r_0, t_6)\}$,$\{(r_1, t_0), (r_1, t_3), (r_1, t_4), (r_1, t_6)\}$, $\{(r_2, t_3)\}$.*

From the example above, one can produce the output set by simply scanning the table one tuple at a time, performing Hamming distance calculation via the XOR operation, and reporting the tuple as an output if the computed Hamming distance is smaller than or equal to $\hat{h}$. If $|S| = n$, then the cost of computing Hamming-select consists of $O(n)$ tuple reads and $O(n)$ Hamming-distance computations. Similarly, the cost of computing Hamming-join between the two datasets R and S, where $|R| = m$ and $|S| = n$ respectively, with a nested-loop join algorithm, consists of $O(mn)$ tuple reads and $O(mn)$ Hamming-distance computations. The focus of this paper is to develop a Hamming-distance-based tree index to reduce the above costs.

[1]Different from the kNN-join, $\hat{h}$-join for datasets R and S is symmetric, i.e., $\hat{h}$-join(R, S)= $\hat{h}$-join(S, R).

Table 2: An example illustrating a Hamming-distance query.

(a) Table S

tuple	**binary** U
t_0	001 001 010
t_1	001 011 101
t_2	011 001 100
t_3	101 001 010
t_4	101 110 110
t_5	101 011 101
t_6	101 101 010
t_7	111 001 100

(b) Table R

tuple	**binary** U
r_0	101 100 010
r_1	101 010 010
r_2	110 000 010

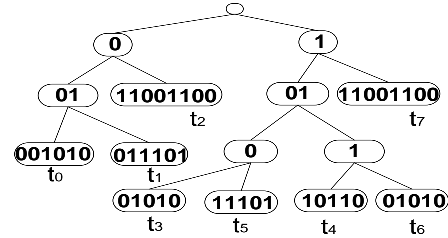


Figure 1: Radix Tree

4. HAMMING-SELECT ALGORITHMS

In this section, we first introduce the basic concept and principles of binary hash codes, and illustrate the Radix-Tree-based approach. We then introduce two variants of our proposed HA-Index, namely the static and dynamic HA-indexes along with their associated algorithms.

4.1 Properties of Binary Codes

DEFINITION 3. *A binary code $\hat{U}$ is said to be a fixed-length substring ($FLSS$) of another binary code U if $|U| = |\hat{U}|$ and there exist i and j, $1 \leq i$, $i + j \leq |U|$ such that $\forall i, i \leq v \leq i + j$, and $U[v]$ = $\hat{U}[v]$. Thus, only the bits between i and $i + j$ are the same and all the remaining can be any combination of* 0*s and* 1*s.*

For example, consider Tuple t_0 in Table 2a. Let $\cdot$ denote a 0 or a 1. Based on the above definition, $\hat{U}$="$\cdot\cdot\cdot\cdot$**0101**$\cdot$" is one $FLSS$ of t_0's binary code "0011**0101**0". Alternatively, $\hat{V}$ ="101$\cdot\cdot\cdot\cdot\cdot\cdot$" is not an $FLSS$ of t_0's binary code.

DEFINITION 4. *A binary code, $\hat{U}$, is the fixed-length SubSequence ($FLSSeq$, for short) of a binary code U if there exists a strictly increasing sequence of indices of U such that $\forall j \in \{1, 2, \ldots, k'\}$, we have $U[j] = \hat{U}[j]$ and $|U| = |\hat{U}|$.*

For example, $\hat{U}$="$\cdot\cdot\cdot$ **0** $\cdot$ **1** $\cdot$ **1** $\cdot$" is one possible $FLSSeq$ of t_0's binary code "001**0**0**1**0**1**0" in Table 2a. Thus, $\hat{U}$ belongs to Set $FLSSeq$ of Tuple t_0. To compute the Hamming distance between an $FLSSeq$ and a query binary code, we only count the bit difference in the corresponding effective bit positions. For instance, if one $FLSSeq$ is $\hat{U}$="$\cdot\cdot\cdot$ **0** $\cdot$ **1** $\cdot$ **1** $\cdot$" and the query binary code is U="001**0**0**1**0**1**0", the Hamming distance $||\hat{U}, U||_h$=2.

PROPOSITION 1. ***Hamming Downward Closure Property*** *A binary code $U \in \hat{h}$-select(t_q, S) iff each $FLSS$ of U, say U_{FLSS}, (each $FLSSeq$ of U, say U_{FLSSeq}, respectively) meets the condition $||t_q, U_{FLSS}||_h \leq \hat{h}$ ($||t_q, U_{FLSSeq}||_h \leq \hat{h}$, respectively).*

We omit the proof for simplicity. Instead, we illustrate the above proposition using the following example.

EXAMPLE 2. *Refer to the Hamming-distance query in Example 1 and Table 2. Suppose that the Hamming-distance threshold $\hat{h} = 2$. Consider the following example cases:*

- *Case 1: Given a query binary code t_q = "110010010", since one FLSS, U_{FLSS} = "$001 \cdots\cdots$", is the binary code of an $FLSS$ for both t_0 and t_1 and $||U_{FLSS}, t_q||_h \geq 3$, then neither t_0 nor t_1 can belong to $\hat{h}$-select(t_q, S).*

- *Case 2: Given a query binary code t_q = "110110010", the binary code "$\cdot$ 11001100" is an FLSS (U_{FLSS}) for both t_2 and t_7, $||U_{FLSS}, t_q||_h \geq 3$, thus, neither t_2 nor t_7 can belong to $\hat{h}$-select(t_q, S).*

- *Case 3: Given a query binary code t_q = "110100010", the binary code "$1010 \cdot 1 \cdots$" is an $FLSSeq$ for both t_3 and t_5, $||U_{FLSSeq}, t_q||_h \geq 3$, therefore, neither t_3 nor t_5 can belong to $\hat{h}$-select(t_q, S).*

4.2 Radix-Tree-Based Approach

The idea behind using a Radix-Tree index (also termed the PATRICIA trie) [19] is to merge the XOR operations for various binary codes if they happen to share $FLSS$s, e.g., similar to Case 1 of the example above. One XOR operation on a common $FLSS$ can be used to verify all participant tuples in this $FLSS$. Thus, we can build a prefix tree out of the binary codes. Based on the above closure property (Proposition 1), we can compute the Hamming distance with prefixes of the Radix-Tree from the root to find qualifying binary codes in a top-down fashion.

EXAMPLE 3. *Figure 1 gives the corresponding Radix-Tree for the binary codes in Table 2. From the Radix-Tree, Tuples t_0 and t_1 in Table 2 share the same FLSS U_{FLSS} = "$001 \cdots\cdots$". Given the query binary code t_q = "110010110" and a Hamming-distance query threshold $\hat{h} = 2$, both Tuples t_0 and t_1 can be discarded without computing the whole Hamming distance for all binary positions, because the Hamming distance from U_{FLSS} with the first three bits of t_q is bigger than the predefined threshold $\hat{h}$. Thus, processing the Hamming-distance-based select can stop early at the upper level of the Radix-Tree.*

Notice that although useful in the above example, the Radix-Tree-based approach has several disadvantages, mainly due to its prefix-sensitiveness. For example, Tuples t_2 and t_7 in Figure 1 are split into two branches in the Radix-Tree, although only the first bit in the two tuples is different while all their remaining bits are the same. Thus, the search path would go to different branches of the tree and redundant computations in these two branches cannot be avoided. In the worst case, if the binary codes in the Radix-Tree do not share common prefixes, then searching from the root will bring the computation cost as bad as $O(2^L)$, because it would go through every branch of the Radix-Tree. As a result, we propose the HA-Index to address the prefix-sensitivity of the Radix-Tree-based approach.

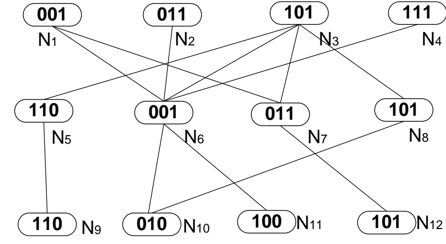


Figure 2: Static HA-Index

4.3 Static HA-Index

The idea behind the Static HA-Index is to share the common substrings, i.e., the maximal $FLSS$s, in contrast to sharing the common prefixes for the binary codes of the underlying dataset. Thus, redundant Hamming distance computations can be avoided. Recall Case 2 of Example 2, the $FLSS$ for t_2 and t_7 is "$\cdot$01101010". For the Radix-Tree-based approach in Figure 1, searching for the qualifying tuples would proceed to different paths, which introduces redundant computations. Thus, if we are able to realize an index that shares the common $FLSS$s, we would be able to avoid redundant and unnecessary Hamming-distance computations.

Static bit segmentation: We segment the binary codes into fixed-length contiguous substrings (called fixed-length segments). For instance, assuming that each segment is of Size 3, the binary code for tuple t_2 is divided into three segments, "011", "001" and "100". The path along these segments can be traced via an undirected path. For example, the path that corresponds to tuple t_2 is illustrated in Figure 2 where it connects Nodes N_2 to N_{11} via Intermediate Node N_6. Meanwhile, the path of Tuple t_7 includes Nodes N_4, N_6 and N_{11}. Thus, Tuples t_2 and t_7 can share the same vertex nodes N_6 and N_{11}. While traversing the index, the Hamming-distance computation for Nodes N_6 and N_{11} will be performed only once. In the next section, we demonstrate how the Static HA-Index can be used to evaluate both the Hamming-select and Hamming-join operations.

The static HA-Index has several limitations though. Both the height and the length of the paths in the Static HA-index are sensitive to the segment size. Because the segment sizes are fixed, it is possible to miss common bit substrings that do not align to segment boundaries. Also, both the Radix-Tree and the static HA-Index optimize for the $FLSS$s of the binary codes. An index that would support $FLSSeq$s, in contrast to just the $FLSS$s (recall that the $FLSS$s are subsets of the $FLSSeq$s), would allow for more shared distance computations and hence additional savings. Consider Case 3 of Example 2. Both the Radix-Tree and Static-HA-Index approaches fail to capture the common $FLSSeq$ between t_3 and t_5. In the next section, we introduce the Dynamic HA-Index to address these limitations.

4.4 Dynamic HA-Index

DEFINITION 5. *Gray Order: is an ordering of the binary codes such that consecutive binary codes differ only by one bit, i.e., the Hamming distance between two consecutive binary codes that are sorted according to the Gray order is equal one [13].*

PROPOSITION 2. ***Gray Order and Clustering***: *When the binary codes are ordered based on the Gray order, data tuples are naturally clustered [20],i.e., the Hamming distance between consecutive ordered binary codes is small as the consecutively ordered binary codes share common $FLSSeq$s.*

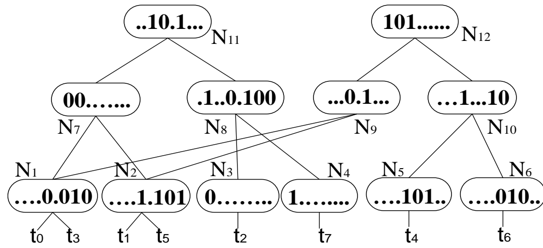


Figure 3: Dynamic HA-Index

For instance, the data tuples in Table 2 can be ordered based on the Gray order of their corresponding binary codes in descending order, and the resulting sorted order is $\{t_0, t_1, t_2, t_7, t_4, t_6, t_3, t_5\}$. Observe that the sorted binary codes provide two important properties, namely the downward closure and the clustering properties, that facilitate efficient Hamming-distance-based query processing. Thus, our aim is to realize an index structure that preserves and leverages these properties. The Dynamic HA-Index will strategically divide the binary codes into segments (i.e., sequences of data points that are close in their binary values according to the Gray order). As such, the clustering property is preserved to ensure that nodes with similar $FLSSeq$s are close to each other in the index. For example, Tuples t_2 and t_7 are ordered next to each other, and these properties can overcome the prefix-sensitivity of the Radix-Tree-based approach.

In the Dynamic HA-Index, the leaf nodes store data tuples while the non-leaf nodes store the $FLSSeq$s of the children nodes. Refer to Figure 3 for an illustration. Internal node N_1 represents the $FLSSeq =$ "$\cdots 0 \cdot 010$" of Tuples t_0 and t_3. Internal node N_2 represents the $FLSSeq =$ "$\cdots 1 \cdot 101$" that is common to both Tuples t_1 and t_5. Furthermore, Internal Node N_7 represents the $FLSSeq$ for Nodes N_1 and N_2. Notice, all the descendants of an HA-Index node can be safely discarded from further Hamming-distance computations if the node's corresponding $FLSSeq$ does not qualify the Hamming-distance threshold, thereby reducing computation overheads.

4.5 Dynamic HA-Index Manipulation

The primary objective of all the Dynamic HA-Index manipulation algorithms, including build, delete, and insert, is to maintain the $FLSSeq$ properties of the index while keeping the size of the index reasonably small.

Bulkloading builds the Dynamic HA-Index in a bottom-up fashion. It has two steps. The first step sorts all the data tuples according to the Gray order of their nondecreasing binary codes. The second step scans these tuples sequentially using a sliding window with w slots to form index nodes. Algorithm 1 illustrates the pseudo-code to build the Dynamic HA-Index. A queue is initialized to store the temporary nodes from the window (Line 2). From the tuples within a window, Function `extractFLSSeq` extracts the maximal $FLSSeq$s from the tuples' binary codes to form new parent nodes (Line 5), and denotes the new binary code of the child node. Then, the new temporal node is inserted into the queue (Line 7). For instance, Tuples t_0 and t_1 share the same $FLSSeq =$ "$0010 \cdot 1 \cdots$". Thus, this $FLSSeq$'s corresponding new node is formed and is inserted into the queue. To save memory storage, Function `extractFLSSeq` captures the binary code of t_0 as "$\cdots \cdot 0 \cdot 010$". Therefore, the non-leaf nodes with the same $FLSSeq$ are consolidated into one node. Hence, Tuple t_3 would be denoted with the same binary code as that of t_0, and would share

Algorithm 1: H-Build

Input: T: Set of data points, w: Window, md: Depth of HA-index, s: Sliding window size
Output: *HA*:HA-Index for dataset T

1 Sort T based on the non-decreasing Gray order of the tuples' binary codes;
2 q: Queue;
3 **for** *each data element* t_i *of* T *inside Window* w **do**
4   var n, $\hat{n}$: Node;
5   n, $\hat{n} \leftarrow$ extractFLSSeq($t_i, \cdots, t_{i+w}$); // n, the parent node of $\hat{n}$
6   **if** $\hat{n}$ *is new* **then**
7   insert $\hat{n}$ into the current level of the HA-Index.
8   **end**
9   **else**
10   update $\hat{n}$'s frequency
11   **end**
12   **if** n *is not empty* **then**
13   q.enqueue(n);
14   **end**
15   **else**
16   put Tuple t_i inside Window w into the top level of the HA-Index;
17   **end**
18   w← w+s; //sliding the window
19 **end**
20 var *d*:0, *begin*:0, *end*:q.size;
21 **while** *q is not empty and* $d \leq md$ **do**
22   // Process similar to Lines 4-18
23   // Use two pointers for q to record the HA-Index depth d
24 **end**

the same binary codes. Notice that we record the frequency of each node (Line 6-11). For example, Node N_1 represents the binary code for t_0 and t_3. Thus, the frequency for N_1 is 2. If tuples inside the window do not share any $FLSSeq$ among each other, these tuples are linked to the top level of the HA-Index (Line 16). The window continues to slide until all the data points are scanned in the first round. Lines 21-24 merge the internal nodes as Lines 4-18 and we can use two pointers `begin` and `end` for the queue to indicate the depth. The building process continues until the desired depth is reached.

In addition, more than one leaf node can be linked to the same internal node, e.g., Tuples t_1 and t_5 are linked to Internal Node N_1 in Figure 3. Thus, we build a hash table for the bottom node, e.g., N_1, where the key is the leaf node's binary codes, and value is the tuple's ID. Naturally, if users only want to learn the qualifying binary codes, then there is no need to keep the leaf nodes of The HA-Index. An HA-Index without leaf nodes could save the overhead of building hash tables, and can be used in MapReduce Hamming-join as in Section 5.

Deletion removes a tuple with its corresponding binary code from a Dynamic HA-Index. Algorithm 2 gives the corresponding process. First, a leaf node that contains the tuple to be deleted is located by depth-first search using the tuple's binary code as the search key. One stack is used to denote the unexplored paths. Function `bitmatch` tests whether one binary code is the $FLSS$ or $FLSSeq$ of the deleted tuple (Lines 3 and 14). Then, the tuple is removed from the HA-Index. After deletion, the frequency of the corresponding node needs to be decremented (Lines 5 and 16). If one node contains 0 or less entries, it is removed.

Inserting a new data tuple into a Dynamic HA-Index is similar to the deletion process. Insertion uses a depth-first search to locate the corresponding leaf node, then the search process looks for the leaf node that shares the maximal $FLSSeq$ with the newly inserted data tuple. If no such leaf node is found, we put the newly inserted data

Algorithm 2: H-Delete

```
Input: t_q: Deleted query tuple, HA: HA-Index for queried dataset
1  s: Stack;
2  for each top level node n_i in HA do
3      if bitmatch(t_q, n_i) then
4          s.push(n_i);
5          n_i.frequency ← n_i.frequency-1 ;
6          remove n_i from HA if n_i.frequency is 0 ;
7      end
8  end
9  while s is not empty do
10     var n: Node;
11     s.pop(n);
12     if n is a non-leaf node then
13         for all child nodes c of n do
14             if bitmatch(t_q, n_i) then
15                 s.push(n_i);
16                 n_i.frequency ← n_i.frequency-1 ;
17                 remove n_i from HA if n_i.frequency is 0 ;
18             end
19         end
20     end
21     else
22         break;
23     end
24 end
```

Algorithm 3: H-Search

```
Input: t_q: Query tuple, ĥ: Hamming distance query threshold, HA:
       HA-Index for queried dataset
Output: ret: Qualified tuple in HA within Hamming distance ĥ from
        tuple t_q
1  q: Queue.
2  for each top level node n_i in HA do
3      if hdist(t_q, n_i) ≤ ĥ then
4          n_i.h ← hdis(t_q, c);
5          q.enqueue(n_i);
6      end
7  end
8  while q is not empty do
9      var n:Node;
10     q.dequeue(n);
11     if n is a non-leaf node then
12         for all children node c of n do
13             if (hdis(t_q, c)+n.h)≤ ĥ then
14                 var m:Node;
15                 m.b ← combine(c.b, n.b); //combine binary code of
                   c and n
16                 m.h ← hdis(t_q, c)+n.h; //update Hamming distance
17                 m.children ← c.children ;
18                 q.enqueue(m);
19             end
20         end
21     end
22     else
23         var binary ← getBinary(n);
24         var tuple ← gettuple(binary);
25         ret.insert(tuple);
26     end
27 end
28 output ret;
```

tuple into a temporary buffer. When the buffer reaches a predefined maximum size, a process similar to H-Build is invoked to append these newly inserted tuples into the existing HA-Index. We omit these details here for brevity as they are similar to Algorithms H-Delete and H-Build.

4.6 HA-Index Query Processing

With the dataset organized in an HA-Index, H-Search traverses the index to visit the relevant index nodes in a breadth-first order with a queue to keep track of the unexplored qualifying paths that match the query's binary code. Algorithm 3 gives the pseudocode for H-Search. Initially, H-Search fetches the index nodes/data points from the top level of the HA-Index (Lines 2-6). If the Hamming distance between the query tuple and the pattern of the corresponding node is smaller than the threshold $\hat{h}$, then the node is inserted into the queue. For the non-top level nodes, in each round, the binary code of a node is examined against the query binary code by invoking a Hamming-distance computation. If its corresponding Hamming distance is smaller than the threshold (Line 12), the node is further explored (Lines 13-17). When a leaf node of the HA-Index is reached, the qualified data tuples are collected and are inserted into ret (Line 23-25). The algorithm terminates when all the entries from the qualifying nodes are examined.

To illustrate the H-Search Algorithm, consider the tuples in Table 2a. Figure 3 gives the corresponding HA-Index. The execution trace is given in Table 3. Suppose that the query binary code is t_q = "010001011" and the Hamming-distance threshold is 3. Initially, the Hamming distance between t_q and the top-level entries, i.e., $||N_{11}, t_q||_h = 1$ and $||N_{12}, t_q||_h = 3$, where both are no bigger than 3. Thus, Nodes N_{11} and N_{12} are pushed into the queue, and ret is still empty. Next, the children nodes of N_{11}, i.e., Nodes N_7 and N_8 are visited. The Hamming distances $||N_7, t_q||_h = 1$ and $||N_8, t_q||_h = 4$ are computed. As a result, the corresponding qualifying binary codes for Nodes N_{11} and N_7 are combined, which results in the pattern "$0010 \cdot 1 \cdots$". Thus, $[N_7, N_{11}]$ is put into the queue. But Node N_8 is discarded from the qualifying candidates because the path $N_{11} \rightarrow N_8$ has a combined Hamming distance $||N_{11}, t_q||_h + ||N_8, t_q||_h > 3$. Then, N_{12} is explored and its children nodes(e.g.,N_9 and N_{10}) are visited. According to the Hamming-distance closure properties, $[N_9, N_{12}]$ is inserted into the queue as well, while N_{10} is discarded. The H-Search process continues until the queue is empty as shown in Table 3. Finally, Tuple t_0 is reported as one output tuple qualifying the query. Notice that each node maintains a *visited* flag to indicate whether the node has already been visited or not. This helps avoid redundant Hamming-distance computations. For example, Nodes N_1 and N_2 are already visited. Therefore, we do not need to compute the Hamming distance for both nodes again, and hence avoid unnecessary distance computation overhead. In addition, Algorithm H-Search for the dynamic HA-Index can be applied to the static HA-Index, and thus is not repeated in the paper.

Table 3: Sample execution trace for H-Search that corresponds to searching the dataset in Table 2a given the query binary code t_q = "010001011"

Queue	Qualified tuples ret
N_{11}, N_{12}	$\emptyset$
$N_{12}, [N_7, N_{11}]$	$\emptyset$
$[N_7, N_{11}], [N_9, N_{12}]$	$\emptyset$
$[N_9, N_{12}]$	t_0
$\emptyset$	t_0

4.7 Analysis

EXAMPLE 4. *Assume that we have eight tuples t_0 = "000", t_1 = "001", t_2 = "010", $\cdots$, and t_7 = "111", where all binary codes are distinct. At most 3 bits are needed to represent all the tuples, i.e., the length L of the hash values is 3. According to the H-Build process with Window Size of 2, the output HA-Index is illustrated in Figure 4.*

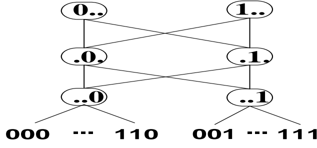


Figure 4: Full binary codes and the corresponding HA-Index

Observe that the number of internal nodes of this HA-index is 6, and the number of edges is 8. Based on the breadth-first-search strategy of the H-Search algorithm, the worst search cost is bounded by the number of internal nodes and the number of edges, denoted by $|V|$ and $|E|$, respectively. Refer to Figure 4 for illustration. The search cost is at worst 14. Suppose that the number of distinct binary codes is n_d, and $n_d = 2^L$. An HA-Index for this example is illustrated in Figure 4. The reason is that the $FLSSeq$ for the binary codes in the same window is maximized with Length $L-1$, and this $FLSSeq$ also shares the maximum similar patterns with its neighboring $FLSSeq$. Therefore, for the dataset with $n_d = 2^L$ data points and the built HA-Index as in Figure 4, the number of internal nodes $|V| = 2L$ or $|V| = 2\log_2 n_d$, and number of edges $|E| = 4(L-1)$ or $|E| = 4(\log_2 n_d - 1)$. This can be proven via induction (Details are omitted for brevity). Thus, the worst case for H-Search on this HA-Index is $|V| + |E| = 2\log_2 n_d + 4(\log_2 n_d - 1)$, i.e., is $O(\log_2 n_d)$. This indicates that H-Search can achieve the best performance under this scenario. We will discuss more general cases later.

Window size We discuss the relationship between window size, say as w, and binary string length L. Inspired by the previous extreme example, it is desirable that the n tuples can span the space of binary strings of L bits. L can be chosen such that $L = \lceil \log_2 n \rceil$, i.e., $2^{L-1} < n \leq 2^L$. Thus, if n is closer to 2^L, then the corresponding HA-Index is closer to the extreme case in our motivating example above. On the other hand, the smallest value for n is $2^{L-1} + 1$, and this is the worst case, i.e., the sparsest distribution of tuples on the space of binary strings of Length L. For the simplicity of discussion, we assume that the hashed binary strings are uniformly distributed.

Under the above assumption, the maximum Hamming distance L_m for a window of size of w satisfies $\lceil \log_2 w \rceil \leq L_m \leq L$. If $L_m = L$, then the binary strings in the same window cannot be merged together since no shared bit position exists. Therefore, a careful choice should be made on the window size w. The extreme case when $w = n$ is apparently a bad choice since no sharing pattern can be extracted from the window. A similar argument applies for $w = 1$. For smaller values of w, many internal nodes are generated and this results in indexes with larger heights. A suggested value for the window size w is $w = 2^{\lceil L/2 \rceil}$ when $n \approx 2^L$. Suppose that $w = 2^{\lceil L/2 \rceil}$, then the maximum Hamming distance L_m in each window satisfies $\lceil L/2 \rceil \leq L_m \leq L$.

Number of nodes in an HA-Index If $n \approx 2^L$, suppose that there are only few windows with $L_m = L$ and we denote the number of these binary codes within that window as δ_1. Since the leaves share about half of the bits in their binary codes, this results in a number of $2^{\lceil L/2 \rceil} + \delta_1$ of internals nodes 1-level higher above the leaves, where $\delta_1 \ll 2^{\lceil L/2 \rceil}$. With the HA-index progressively growing, a higher level with $2^{\lceil L/4 \rceil} + \delta_2$ internal nodes can be built where $\delta_2 \ll \delta_1$. In the same way, the HA-index grows to the highest level with $2^{\lceil L/2^h \rceil} + \delta_h$ uppermost internal nodes, where h is the height of the index. Thus, the total number of nodes $|V|$ in the HA-index can be estimated by:

$$
\begin{aligned}
|V| &= 2^{\lceil L/2 \rceil} + 2^{\lceil L/4 \rceil} + \cdots + 2^{\lceil L/2^h \rceil} + \sum_{i=1}^{h} \delta_i \\
&= 2^{\lceil \log_2 n^{1/2} \rceil} + 2^{\lceil \log_2 n^{1/4} \rceil} + \cdots + \sum_{i=1}^{h} \delta_i \\
&< 2 \times 2^{\lceil \log_2 n^{1/2} \rceil} + \sum_{i=1}^{h} \delta_i \\
&< 2 \times 2^{\lceil \log_2 n^{1/2} \rceil} \\
&= O(\sqrt{n}).
\end{aligned}
$$

We can safely ignore the delta part since the summation is negligible compared to the dominant term.

If $n \approx 2^{L-1}$, then the window size w needs to shrink to a proper length. Based on the assumption of uniform distribution of the binary strings and Gray ordering, a proper window size can be set to $w = 2^{\lceil L/4 \rceil}$. The maximum Hamming distance L_m within a window satisfies $\lceil L/4 \rceil \leq L_m \leq L$. A similar analysis suggests that the number of internal nodes $|V'|$ satisfies:

$$
\begin{aligned}
|V'| &= 2^{\lceil L/4 \rceil} + 2^{\lceil L/4^2 \rceil} + \cdots + 2^{\lceil L/4^h \rceil} + \sum_{i=1}^{h} \delta_i' \\
&= 2^{\lceil \log_2 n^{1/4} \rceil} + 2^{\lceil \log_2 n^{1/4^2} \rceil} + \cdots + 2^{\lceil \log_2 n^{1/4^h} \rceil} + \sum_{i=1}^{h} \delta_i' \\
&= O(\sqrt[4]{n}).
\end{aligned}
$$

Number of Edges in an HA-Index For the number of edges in an HA-index, there are two extreme cases. Suppose that $n \approx 2^L$ and we have already discussed that the two levels above the leaves contain $2^{\lceil L/2 \rceil}$ and $2^{\lceil L/4 \rceil}$ internal nodes, respectively. The worst case is that each of the $2^{\lceil L/2 \rceil}$ nodes connects to each of the $2^{\lceil L/4 \rceil}$ nodes. This induces about $2^{3L/4}$ edges. Similarly, the edge number can be estimated,

$$
|E| = 2^{3L/4} + 2^{3L/8} + \cdots + 2^{3L/2^{h+1}} < 2 \times 2^{3L/4} = O(\sqrt[4]{n^3}).
$$

On the other hand, the best estimate is that there are no cross edges between the children and different parents. For this case, a lower bound of the number of edges is $O(\sqrt{n})$, which is similar to the number of vertices.

Query Cost and Storage Space of the HA-Index The cost of H-Search is bounded by the number of nodes and edges, i.e., $|V| + |E|$. Therefore, the worst cost for H-Search is traversing all the edges and nodes in the HA-index. This indicates that H-Search can be bounded in the range $[O(\sqrt{n}), O(\sqrt[4]{n^3})]$. Meanwhile, besides the storage of the leaf nodes, the space usage of the HA-index also depends on the sum of the number of nodes and edges, i.e., $[O(\sqrt{n}), O(\sqrt[4]{n^3})]$. Compared to the state-of-the-art approaches [4, 8], the HA-Index does not need to maintain several copies of the dataset. Thus, it can be kept in memory for fast query processing. Furthermore, the internal nodes of the HA-Index store enough binary information for the whole dataset, and hence introduce low overhead to broadcast an HA-Index to each server.

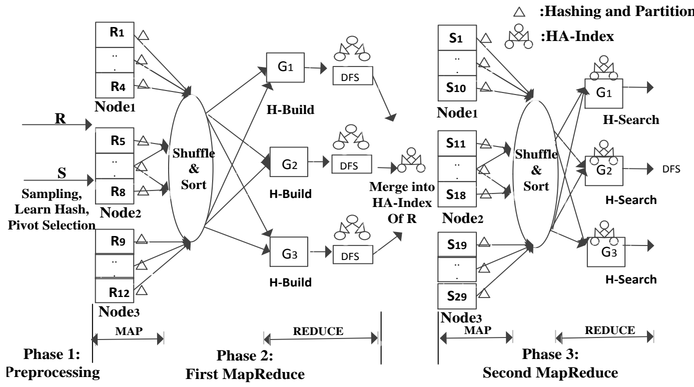


Figure 5: An overview of Hamming-join processing in MapReduce.

5. PARALLEL ALGORITHM FOR HAMMING-JOIN

To process Hamming-join on two datasets, say R and S, one straightforward approach is to build an HA-Index for R, then execute H-Search on the built index for each tuple of S. However, to build an HA-Index for R, sorting R would be slower as R gets larger. Secondly, executing H-Search between each tuple of S and the HA-Index for R would make the query time bounded by the number tuples in S. In this section, we address these limitations of the centralized environment and introduce Hamming-join on the MapReduce platform [21].

To support Hamming-join over MapReduce, we focus on two important issues. First, load balancing is important because the slowest mapper or reducer determines the job running time. Secondly, data shuffle from the mappers to reducers usually results in large disk I/O and network communication costs that heavily influences the run-time performance. Therefore, we not only need to reduce the data shuffle cost, but also make sure data partitions in each mapper or reducer are well balanced.

5.1 Overview of MapReduce-based Hamming-join

In this section, we introduce our implementation of the Hamming-join operation in MapReduce. As Figure 5 illustrates, the proposed algorithm includes three phases as explained below.

- **Preprocessing phase** Retrieve a sample from Datasets R and S. Then, use the sampled data to learn the hash function H. To handle data skew, build a data histogram for the sampled data and learn the data partitioning rule for the entire MapReduce job.
- **Global HA-Index building phase** Assume that the size of R is smaller than that of S. Partition R based on the pivot values from the data preprocessing step, then build the HA-Index for each partition using MapReduce by calling the H-Build function. Then, merge each local HA-Index to realize a global HA-Index for R.
- **Hamming-join phase** To join HA-Index of R with tuples in S, two possible options are applicable based on the size of R. More details are given later.

To learn the hash function, we utilize a random sample obtained from both R and S using reservoir sampling [22]. With the learned hash function H, high-dimensional data tuples in R and S are mapped into their corresponding binary codes. As discussed in the previous section, hash binary codes are ordered using the Gray order to preserve the clustering property. Hence, the data in each partition is more likely to share common $FLSSeq$ patterns. Then, we build the data histogram for the binary codes of the sampled data, and get a set of pivot values, denoted by Pv, for each Partition Pt_m. This guarantees that each partition receives approximately the same amount of data, where data in the various partitions is ordered according to the Gray order. More formally, given a set of data partitions Pt, and a set Pv of corresponding binary code values that form the partitioning pivots, Tuple $t_i \in Pt_m$, if the Gray order for t_i's binary code, say $\hat{U}_i$, belongs to the pivot range, i.e., $\hat{U}_i \in [Pv_m, Pv_{m+1})$, where Pv_m and Pv_{m+1} are the pivot values for Partition Pt_m.

Thus, let $|Pt_m|$ be the number of tuples belonging to Partition Pt_m, Pivot set Pv partition dataset R, s.t $R = \bigcup_{m=1}^{N} Pt_m$, and $|Pt_m| \simeq |Pt_{m+1}|$. Therefore, we can build the HA-Index and Hamming-join in each server as illustrated below.

5.2 Global HA-Index Building

Given the set of pivot values Pv selected in the preprocessing step, a MapReduce job partitions the data and builds an HA-Index locally in each partition. Specifically, before launching the map function, the selected pivots Pv and the learned hash function H are loaded into memory in each mapper via distributed cache in MapReduce. A mapper sequentially reads each input data tuple, say t_i, from the mapper's corresponding partition. The hash function maps the high-dimensional input data tuples into their corresponding binary codes, i.e., U. Then, a binary search is performed for the closest pivots in Pv. For the closest partition region, Partition ID is assigned. Finally, the mapper(s) produce(s) as output each object t_i along with its Partition ID, original dataset tuple identifier (R or S), and its binary code value U.

In the data shuffling phase, the key-value pairs emitted by all map functions are grouped by each distinct Partition ID, and a reduce function is called within each node. Each reduce function computes the local HA-Index via the H-Build function of Section 4, and produces the local HA-Index as output. In addition, a post-processing step to merge the various local HA-Indexes into one global HA-Index. Mainly, non-leaf nodes with the same $FLSSeq$ from the different local HA-Indexes are merged into one node, and the corresponding edges between the index nodes are relinked. Because the HA-Index is relatively small, the processing overhead is acceptable. After the first MapReduce job finishes, the global HA-Index for dataset R is built. This index is used by H-Search in the next phase.

5.3 Hamming-join

The second MapReduce job performs the Hamming-join in two possible ways.

Option(A): When Dataset R is small, i.e., storage of the leaf nodes of the HA-Index does not dominate the space of the HA-Index, the HA-Index maintains the leaf nodes as in Figure 3. Next, the Map function partitions Dataset S into N parts, i.e., $S = \bigcup_{i=1}^{N} S_i$. Then, it duplicates the global HA-Index for Dataset R and broadcasts to each server. The Map function computes the Hamming-join for Partition S_i and the replicated HA-Index of R. Specifically, before launching the MapReduce Job, the master node broadcasts the pivots Pv, the hash function H, and the global HA-Index of R to various servers. The main task of the mapper in the

second MapReduce Job is to map high-dimensional data into binary codes, then partition dataset S into N partitions. Next, each reducer performs the Hamming-join between a pair of HA-Index and $\hat{S}_i$, and output the Hamming-join results.

Option(B): If Dataset R is big, e.g.,the number of tuples $|R|$ is more than millions, the storage of leaf nodes of the HA-Index dominates the space usage of the HA-Index. Therefore, the HA-Index of Dataset R does not maintain leaf nodes, and is duplicated to each server. By this way, the H-Search Algorithm 3 only returns the qualifying binary codes for Hamming-select, and a post-precessing step is carried out to find the tuple IDs for the qualifying binary codes. Take query tuple t_6 in Table 2a as an example. The H-Search algorithm computes binary codes from Table 2b, i.e., “101100010" and “101010010", which have a Hamming distance of 3 from t_6. In order to find the tuple IDs for those qualifying binaries, one post-processing step is invoked. Naturally, if Dataset R fits into memory, then the qualifying binaries are joined with R's hash table in memory. On the other hand, if Dataset R is too large to fit in memory, MapReduce hash-join [23] for Dataset R and the qualifying binaries is applied.

5.4 Shuffle Cost Analysis

The performance of MapReduce Hamming-join depends on the running time of Hamming-select as well as on the data shuffling cost. Let $|R| = m$ and $|S| = n$, respectively, d be the data dimension, and N be the number of partitions. In the previous work [4], Dataset R is duplicated and broadcast to each server, and the data shuffling cost is approximate to $O(mNd + nd)$. In this work, instead of duplicating the whole dataset R, only the HA-Index, is broadcast to each server. Hence, the data shuffling cost is reduced to $O(|HA|N+n)$, where $|HA|$ is the size of the HA-index. As introduced in Section 4, the space storage of HA is bound to $[O(\sqrt{m}), O(\sqrt[4]{m^3})]$. Therefore, the shuffling cost is bounded in $[O(\sqrt{m}N + n), O(\sqrt[m]{n^3}N + n)]$.

6. PERFORMANCE EVALUATION

We implement all the algorithms in Java. The experiments for Hamming-select are performed on an Intel(R) Xeon (R) E5320 1.86 GHz 4-core processor with 8G memory running Linux. The experiments on MapReduce are performed on a cluster of 16 nodes of Intel(R) Xeon (R) E5320 1.86 GHz 4-core machines with 8GB of main memory running Linux. We use Hadoop 0.22 and apply the default cluster environment setting. We evaluate the performance of the proposed techniques using the following three high-dimensional real datasets: (1) NUS-WIDE[2] is a web image dataset containing 269,648 images. We use 225-D block-wise color moments as the image features, thus obtaining a 225-dimension data. (2) Flickr[3] is a an image hosting website. We crawled 1 million images and extracted 512 features via the GIST Descriptor [24] (the data dimension is 512). (3) DBPedia[4] data aims to extract structured content from Wikipedia. We extract 1 million documents, and then apply standard NLP techniques to pre-process the documents, e.g., to remove stop words. We use the Latent Dirichlet Allocation (LDA) [25] model to extract topics, and we keep 250 topics for each document.

To evaluate the performance on larger data sizes, we synthetically generate more data while maintaining the same distribution as the original data distribution, e.g., as in [9, 10]. Suppose that the original dataset D has k dimensions. First, we get the frequencies of values in each dimension, and then sort the data in ascending order of their frequencies. Therefore, k copies of the dataset D are generated, one copy per dimension, e.g., D_j one copy of the dataset that is sorted based on the j-th dimension. Then, for each tuple, say t, in Dataset D, $t \in D$, we create a new tuple, say $\hat{t}$, according to the position of each component of t in the corresponding sorted copy D_j. For example, $t = (t_1, \ldots, t_j, \ldots, t_d)$ and t'_j is the first value larger than t_j in copy D_j, then $\hat{t} = (t'_1, \ldots, t'_j, \ldots, t'_d)$. If t_j is the largest element in Copy D_j, then $\hat{t}_j = t_j$. We use“$\times s$" to denote the increase in dataset size, where $s \in [5, 25]$ is the increase or scale factor. We consider the following approaches to evaluate Hamming-select:

(1) **Nested-Loops** is the naive approach to linearly XOR and count the binary data to perform the Hamming-distance computation. (2) **MultiHashTable** [4] is the state-of-the-art to search binary codes for similarity hashing that uses multiple-hash tables to reduce the linear search cost. While a large number of hash tables can achieve better performance, we limit ourselves to just 4 and 10 hash tables to avoid memory overflow. For short, we refer to these two possibilities, as MH-4 and MH-10. (3) **HEngine^s** [8] is the most recent work to improve the MultiHashTable approach in query time and memory usage. (4) **Radix-Tree** is the approach introduced in Section 4.2. (5) **Static HA-Index (SHA-Index) and Dynamic HA-Index (DHA-Index)** are the approaches introduced in Sections 4.3 and 4.4, respectively. SHA-Index(32) or DHA-Index(32) means that the length of the binary code is 32 bits.

We further evaluate the following approaches for kNN-select, and show how the approximate kNN-select can benefit from the enhancement of HA-Index searching over binary codes: (1) **Locality-Sensitive Hashing(E2LSH)** [18] is the state-of-the-art implementation for the data-independent LSH. We use 20 hash tables for E2LSH. (2) **LSB-TREE** [26] uses the Z-order curve to map high-dimensional data into one-dimensional Z-values, and index the Z-values using a B-tree. In our experiments, we build the LSB-Tree with 25 trees to compare the performance.

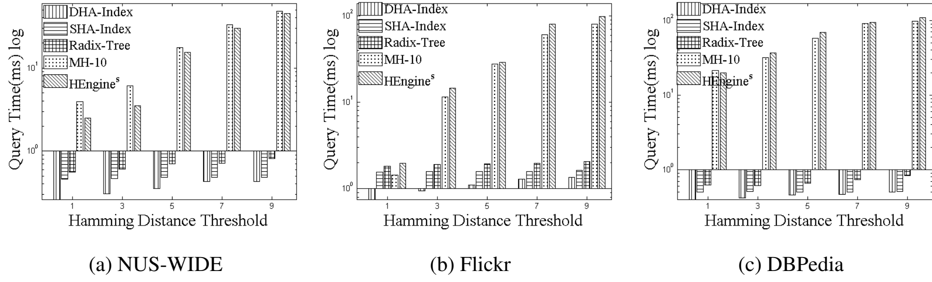
Also, we evaluate the following approaches to test the Self-Hamming-join, and verify how our approach of Map-Reduce Hamming-join can speedup the state-of-art algorithm for exact Self-kNN-join: (1) **Parallel-exact-KNN-join** (short as PGBJ) [10] is the state-of-the-art approach for performing exact kNN-join over multi-dimensional data in MapReduce, and it is 10 times speedup over the Z-order curve based approach [11]. We get the implementation generously provided by the authors [10]. (2) **Parallel Hamming-join via MultiHashTable** (PMH, for short) that handles approximate batch queries for web page duplicate identification [4]. PMH-10 means that 10 hash tables are used. (3) **Parallel Hamming-join via Dynamic HA-Index** (MRHA-Index, for short) is the approach introduced in Section 5. Specifically, in terms of the Hamming-join phase, if Option A is used, we term it MRHA-Index-A, and if Option B is used, we term it MRHA-Index-B.

The performance measures for each algorithm include the query time, the index update time, the index building time, memory usage, and the data shuffling cost. All performance measures are averaged over eight runs. Some running times are not plotted because they would use more than five hours. Unless mentioned, the default value of k is 50, and the Hamming-distance threshold $\hat{h}$ is 3. We choose the state-of-the-art Spectral Hashing [2] as the hash function in our experiments, but our approach is not limited to this hash function.

6.1 Results for Hamming-select

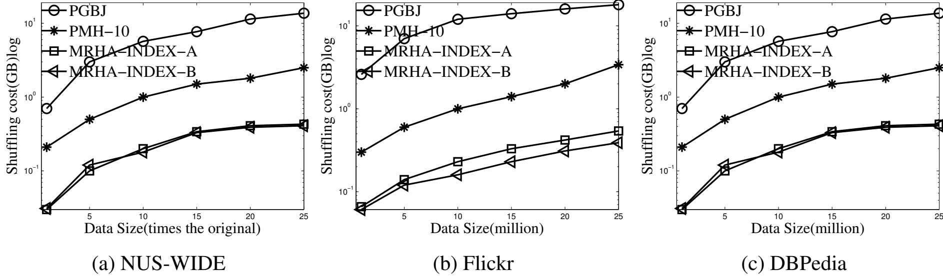
6.1.1 *Effectiveness of the HA-Index*

[2]http://lms.comp.nus.edu.sg/research/NUS-WIDE.htm
[3]htttp://www.flickr.com
[4]http://wiki.dbpedia.org/About



(a) NUS-WIDE (b) Flickr (c) DBPedia

Figure 6: Effect of the Hamming-distance threshold on Hamming select.



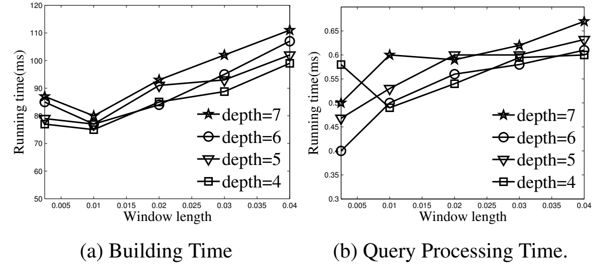
(a) NUS-WIDE (b) Flickr (c) DBPedia

Figure 7: Shuffling cost of Hamming-join and kNN-join.

Table 4 summarizes the query time, index update time, and memory space usage by the various approaches. Specifically, index update corresponds to the operation to delete one tuple first, then insert the same tuple back into the index. From Table 4, we have the following observations: 1) The Radix-Tree and HA-index-based approaches outperform the naive nested-loop and state-of-the-art methods [4, 8] on query time for the three datasets, mainly because the new proposed approach avoids many redundant Hamming-distance computations, and avoids scanning all the underlying data when they are hashed into the same bucket; 2) The HA-Index-based approach, i.e., the Static and Dynamic HA-Indexes, outperforms the Radix-Tree approach. The speedup is around 10 times because the Radix-Tree behaves as a prefix tree when many of the binary codes do not share long common prefixes, and hence cannot avoid the redundant Hamming distance computations; 3) The Static HA-Index shows better index-update time than that of the Dynamic HA-Index because the static segmentation enables us to track different binary segmentations directly, thus, we can search the paths of binary codes more efficiently; 4) The Radix-Tree and the HA-Index-based approaches save more memory than the state-of-the-art methods [4, 8] because the HA-Index-based approaches do not need to duplicate tuples and can share common $FLSS$s and $FLSSeq$s for different binary codes. This can reduce memory usage further; 5) For the Dynamic-HA-Index, if only the internal nodes of the HA-Index are kept, the memory usage can be reduced further. For instance, the memory usage for the Flickr and DBpedia datasets is reduced from 251MB and 225MB to 63MB and 47MB, respectively.

6.1.2 Effect of Hamming-Distance Threshold

We evaluate whether the running time of proposed approach is sensitive to the query threshold $\hat{h}$. Figure 6 gives the data query time when varying the Hamming-distance threshold. Notice that the query time of both the HA-Index-based approaches increases relatively slowly as the threshold increases. The reason is that the searching process in the HA-Index usually terminates early in the



(a) Building Time (b) Query Processing Time.

Figure 8: DHA-Index building time and query processing when varying the window size.

upper-level nodes, and this can improve the query speed. On the other hand, the searching path length of the Radix-Tree is not under control, and it tends to reach each leaf node when the Radix-Tree shares very little and changes to a prefix-tree-like format. However, state-of-the-art methods [4, 8] are sensitive to the Hamming-distance threshold because both approaches have to scan intermediate data to filter out non-qualifying tuples. Hence, the bigger $\hat{h}$ is, the more intermediate results that need to be scanned. This directly degrades the performance.

6.1.3 Effect of HA-Index Parameters

We study the effects of the window length and the index depth of the dynamic HA-Index w.r.t. the index building and query processing times. The window length is normalized by the number of tuples in the dataset. Figure 8a illustrates that the building time for the HA-index drops as the depth decreases. The reason is that index construction stops early while the depth is small. Meanwhile, the HA-Index building time grows as the window size increases because the time to extract the same subpatterns for binaries of one window depends on the number of tuples inside the window. Meanwhile, the query processing time demonstrates stable growth as the

Table 4: Overall comparative study for Hamming-select: The dynamic-HA-Index is the most efficient in terms of query time and space usage, the binary code length is 32 bits. Notice for DHA-Index, **28/11** means 28MB and 11MB space usage for internal and leaf nodes were kept or only internal nodes, respectively.

	(a) NUS-WIDE			(b) Flickr			(c) DBPedia		
method	query time(ms)	update time(ms)	space usage	query time(ms)	update time(ms)	space usage	query time(ms)	update time(ms)	space usage
Nested-Loops	16.42	15.22	/	42.97	41.19	/	59.16	53.53	/
MH-4	6.22	0.21	475	16.09	0.60	712	40.28	0.45	819
MH-10	4.91	0.25	531	14.03	0.83	1204	34.46	0.64	1364
HEngine^s	3.53	0.45	210	14.75	1.14	820	36.91	1.91	763
Radix Tree	1.61	0.19	39	3.98	0.64	365	17.64	0.44	352
SHA-Index	0.87	**0.16**	29	1.75	**0.52**	254	3.54	**0.43**	239
DHA-Index	**0.68**	0.18	**28/11**	**0.74**	0.58	**251/63**	**1.07**	0.51	**225/47**

window size and index depth increase. Observe that the window size increases four times and the query processing time only grows by less than 10%. Thus, the HA-Index is not sensitive to these parameters.

6.1.4 *Comparison of Approaches for kNN-Select*

As introduced in Section 2, Hamming-select is a core operation for evaluating approximate kNN-select. In this section, we demonstrate the performance gains when using the HA-Index to speedup approximate kNN-select. Table 5 illustrates the runtime for data querying and index construction for LSH, LSB-Tree, and the HA-Index-based approaches. Observe that the HA-Index-based approach outperforms the state-of-the-art methods on all tasks when the binary code length is relatively large (i.e., 32 or 64 bits). Compared to the LSH approach, both HA-index-based approaches achieve two orders of magnitude speedup. The reason is that the LSH approach assumes uniformity in the distribution of the underlying data while real datasets are not uniform. In addition, the LSB-Tree can improve the query time compared to the LSH approach. However, the time to build the LSB-Tree index is expensive (more than 24 hours). In addition, the query and index building times for the HA-Index-based approach increases relatively smoothly as the binary code length increases. This demonstrates that the HA-Index approach is robust with the binary code length. Finally, the LSB-Tree consumes extensive disk space to store the index, LSB-Tree uses more than 20GB to store the index for the Flickr data, while the HA-Index-based approach only takes less than 300MB. This significantly reduces disk I/O time for the HA-Index-based approach.

Table 5: Comparison with the state-of-the-art kNN-select approaches, when the dataset size is set to 300k tuples.

Dataset	Algorithm	Query time(ms)	Index build time
	LSH	2400	680(s)
	LSB-Tree(25)	47	37(Hr)
NUS-WIDE	SHA-Index(32)	2.74	68(s)
	SHA-Index(64)	4.78	97(s)
	DHA-Index(32)	1.64	87(s)
	DHA-Index(64)	2.43	103(s)
	LSH	340	1080(s)
	LSB-Tree(25)	63	50(Hr)
Flickr	SHA-Index(32)	2.21	176(s)
	SHA-Index(64)	3.54	189(s)
	DHA-Index(32)	2.17	210(s)
	DHA-Index(64)	2.88	244(s)
	LSH	266	340(s)
	LSB-Tree(25)	59	44(Hr)
DBpedia	SHA-Index(32)	2.94	150(s)
	SHA-Index(64)	4.88	290(s)
	DHA-Index(32)	2.18	230(s)
	DHA-Index(64)	3.85	310(s)

6.2 Results of Hamming-join in MapReduce.

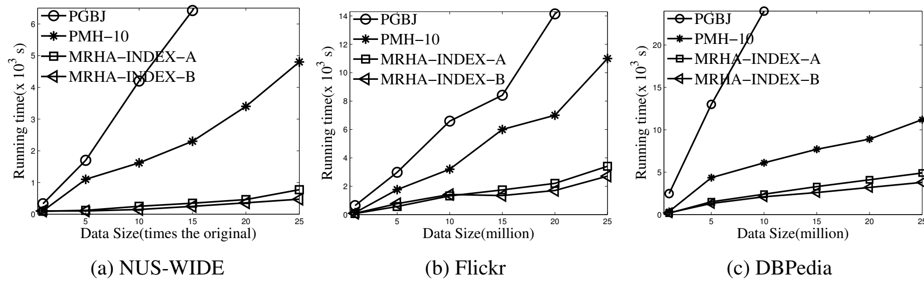
6.2.1 *Shuffling Cost*

We measure the effect of data size on the shuffling cost for PGBJ, PMH and the MRHA-Index. Figure 7 gives the data shuffle costs when the data size varies. The shuffle cost is plotted in logarithmic scale. The smaller the shuffle costs, the better the performance is. We observe that the shuffle costs for approximate kNN-join approach, i.e., PMH and MRHA-INDEX, are 10 times smaller when compared to the PGBJ approach. The reason is that the hashing technique maps the high-dimensional data into binary codes, and hence the data shuffling cost does not depend on the dimensions of the data. Notice that the data shuffling cost for PGBJ increases linearly with the data size. This is two orders of magnitude worse when compared to the data shuffling cost for the MRHA-INDEX approach. Duplicating and distributing the HA-Index into different nodes can improve the data shuffle cost 10 times less than that of the PMH approach. On the other hand, the larger shuffle cost would stop the PGBJ approach from achieving a linear speedup and its corresponding execution time shows quadratic increase. The corresponding running times are given below. Finally, for the Hamming-join step in the HA-Index-based approach, Option B saves more data shuffling cost than Option A because the former does not need to duplicate the whole dataset into each server, and hence the space usage of the HA-Index remains relatively small.

6.2.2 *Scalability and Speedup*

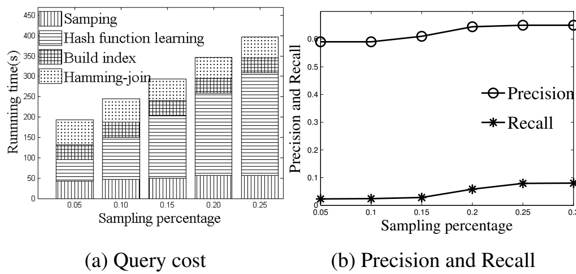
We investigate the scalability of the three approaches in Figure 9. The figure presents the results by varying the data size from 1 to 25 times of the original dataset sizes. From the figure, the overall execution time of PGBJ shows quadratic increase when the data size increases. For example, PGBJ's running time is almost 13 hours when the data is DBPedia$\times$15, which is excessively slow. The approximate kNN-join via similarity hashing always outperforms the PGBJ approach. Comparing with the state-of-the-art PMH-10 approach, the running time of the HA-Index outperforms PMH-10 by 5 times.

6.2.3 *Effect of Data Sampling*



(a) NUS-WIDE (b) Flickr (c) DBPedia

Figure 9: Speedup and scalability: Running time of Mapreduce Hamming-join and kNN-join.



(a) Query cost (b) Precision and Recall

Figure 10: Effect of sampling on query processing time, and precision/recall when varying the sampling data size.

Figure 10a gives the query execution time for the various processing phases of Hamming-join. From the Figure, more sampling of the data reflects the global data distribution more clearly, and this helps the sampling data pivot to partition different regions more evenly, and hence, improves the parallel HA-Index building and Hamming-join query time. The hash function learning usually takes more time, but for real-world applications, we only need to learn the hash function again when a certain amount of the new data is updated, which can save the time. Figure 10b illustrates how data sampling affects the query quality. Observe that the precision and recall can moderately improve as the sampling data size increases. However, the recall value is low.

7. CONCLUDING REMARKS

In this paper, we study the problem of efficiently performing the Hamming-select and Hamming-join operations. The proposed HA-Index approach executes the Hamming-distance-based similarity operations while avoiding unnecessary Hamming-distance computations. Extensive experiments using real datasets demonstrate that the proposed approaches outperforms the state-of-the-art techniques by two orders of magnitude. In future, it would be interest to explore hamming-distance similarity operation for relational operation i.e.,intersection [27].

8. REFERENCES

[1] J. Song, Y. Yang, Y. Yang, Z. Huang, and H. T. Shen, "Inter-media hashing for large-scale retrieval from heterogeneous data sources," ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 785–796.

[2] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing," in *NIPS'08*, 2008, pp. 1753–1760.

[3] M. M. Bronstein, E. M. Bronstein, F. Michel, and N. Paragios, "Data fusion through crossmodality metric learning using similaritysensitive hashing," in *in Proc. CVPR*, 2010.

[4] G. S. Manku, A. Jain, and A. Das Sarma, "Detecting near-duplicates for web crawling," ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 141–150.

[5] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02. New York, NY, USA: ACM, 2002, pp. 380–388.

[6] M. Marvin and A. P. Seymour, "Perceptrons," *MIT Press*, 1969.

[7] D. Greene, M. Parnas, and F. Yao, "Multi-index hashing for information retrieval," in *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, Nov 1994, pp. 722–731.

[8] A. Liu, K. Shen, and E. Torng, "Large scale hamming distance query processing," in *2011 IEEE 27th International Conference on Data Engineering (ICDE)*, April 2011, pp. 553–564.

[9] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using mapreduce," ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 495–506.

[10] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 1016–1027, Jun. 2012.

[11] C. Zhang, F. Li, and J. Jestes, "Efficient parallel knn joins for large data in mapreduce," ser. EDBT '12. New York, NY, USA: ACM, 2012, pp. 38–49.

[12] H. Kllapi, B. Harb, and C. Yu, "Near neighbor join," in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, March 2014, pp. 1120–1131.

[13] F. Gray, "Pulse code communication," in *U.S. Patent 2,632,058*, 1953.

[14] X. Zhang, J. Qin, W. Wang, Y. Sun, and J. Lu, "Hmsearch: An efficient hamming distance query processing algorithm," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, ser. SSDBM. New York, NY, USA: ACM, 2013, pp. 19:1–19:12.

[15] Y. N. Silva, W. G. Aref, P.-Å. Larson, S. Pearson, and M. H. Ali, "Similarity queries: their conceptual evaluation, transformations, and processing," *VLDB J.*, vol. 22, no. 3, pp. 395–420, 2013.

[16] C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish, "Indexing the distance: An efficient method to knn processing," ser. VLDB '01, San Francisco, CA, 2001, pp. 421–430.

[17] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," ser. VLDB '98, San Francisco, CA, 1998, pp. 194–205.

[18] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, no. 1, pp. 117–122, Jan. 2008.

[19] D. R. Morrison, "Patricia;practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 514–534, Oct. 1968.

[20] C. Faloutsos, "Multiattribute hashing using gray codes," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '86. ACM, 1986, pp. 227–238.

[21] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[22] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, Mar. 1985.

[23] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce," ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 975–986.

[24] A. Oliva and A. Torralba, "Modeling the shape of the scene: A holistic representation of the spatial envelope," *International Journal of Computer Vision*, vol. 42, pp. 145–175, 2001.

[25] A. K. McCallum, "Mallet: A machine learning for language toolkit," 2002, http://mallet.cs.umass.edu.

[26] Y. Tao, K. Yi, C. Sheng, and P. Kalnis, "Efficient and accurate nearest neighbor and closest pair search in high-dimensional space," *ACM Trans. Database Syst.*, vol. 35, no. 3, pp. 20:1–20:46, Jul. 2010.

[27] W. J. A. Marri, Q. M. Malluhi, M. Ouzzani, M. Tang, and W. G. Aref, "The similarity-aware relational intersect database operator," in *7th International Conference Similarity Search and Applications, SISAP*, 2014, pp. 164–175.