A Study on Efficient and Secure Set Similarity Joins

March 2018

Mateus Silqueira Hickson Cruz

A Study on Efficient and Secure Set Similarity Joins

Graduate School of Systems and Information Engineering

University of Tsukuba

March 2018

Mateus Silqueira Hickson Cruz

Abstract

Similarity joins are common database operations that can relate records based on their similarity degree. Set similarity joins constitute a specific type of similarity joins that assume records to be sets. Although seemingly restrictive at first, a wide range of data types can be treated as sets during its processing. For example, market basket data can be seen as a set of purchased items, text documents can be sets whose elements are words, and images can be represented as set of chosen features. Due to its flexibility in finding similar data items, set similarity joins have a large number of applications, which include data cleaning, entity recognition and duplicate elimination.

Although a number of works have been proposed to enhance set similarity joins, the development of recent technologies (e.g., many-core processors and fully homomorphic cryptosystems) offers new ways for improvement. In this dissertation, we investigate the use of such emergent technologies, particularly focusing their use on the performance and security aspects of set similarity joins.

Regarding performance, we propose a new scheme of set similarity joins that use graphic processing units (GPUs) to accelerate the computation. As for the security facet, we propose a privacy-preserving scheme to execute two-party set similarity joins using encrypted data.

We highlight the main contributions of this study as follows: (1) A novel method to accelerate the processing of set similarity joins using the massive parallelism provided by GPUs. To overcome the memory limitations of GPUs, we employ a dimension reduction technique, MinHash, and create compact representations of sets. An extensive experimental evaluation shows speedups of more than two orders of magnitude when compared to serial execution. (2) A two-party protocol for secure execution of similarity joins that uses a fully homomorphic cryptosystem, offering security and privacy-preservation when calculating similarity values over encrypted data. We take advantage of the threshold Tversky index to determine whether two sets are similar without disclosing the similarity value itself. To compensate for execution costs incurred by fully homomorphic schemes, we explore CPU parallelization and the adaptation of filter techniques to the context of encrypted data.

Considering the growing amount of data generated by different sources, including sensitive applications, we expect our contributions to aid in the processing of large datasets in a reliable and secure way.

Acknowledgements

This work would not have been possible without the help and support of several individuals.

First, I am grateful for the patience and for the guidance received from my advisor, Professor Hiroyuki Kitagawa, and from my co-advisor, Professor Toshiyuki Amagasa. It was a very enriching experience to work with them.

I also would like to thank my committee members, Professor Miyuki Nakano, Professor Daisuke Takahashi and Professor Jun Sakuma. Their insightful comments and suggestions greatly contributed to improve this work.

I am indebted to the faculty, staff and colleagues of the University of Tsukuba, especially Associate Professor Hideyuki Kawashima, Assistant Professor Chiemi Watanabe, Assistant Professor Yasuhiro Hayase, Assistant Professor Hiroaki Shiokawa, Assistant Professor Claus Aranha, Assistant Professor Neil Millar, Dr. Takahiro Komamizu, Dr. Savong Bou, Wenjie Lu, Yuyang Dong, Rei Funaki, Yumiko Hisamatsu, Tetsuko Sato.

This project was assisted by the Monbukagakushou scholarship, provided by the Ministry of Education, Culture, Sports, Science and Technology of Japan.

Finally, I want to express my gratitude to my family and friends, who encouraged me no matter the distance, and to Lisander, for her love and understanding.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

With the dissemination of technological advances like online social networks and the Internet of Things, we have been witnessing an unprecedented amount of data being generated at a continuously increasing speed. This phenomenon has created numerous opportunities that have been explored by different parties, from governments to enterprises. For instance, due to the volume of data available, it is possible to study epidemic patterns and target potential customers with much more accuracy than before.

Notwithstanding its promises, there are also several challenges inherent to manipulating large data; e.g., efficient processing, adequate storage and responsible management. In particular, because such data is usually generated by multiple sources, there is a growing need for its proper integration in order to explore its full potential. Examples regarding the need for data integration include research collaboration between different institutes, enterprise merging and international law enforcement.

In this context, a proper processing has great impact on the quality of the analysis and on the general use of the data being handled. Considering the case of data stored in databases, a common way to process data that come from different sources is to use the *join* operator, which associates database records based on a specified condition. The join operator is one of the most useful and most explored tools for data processing, but, in its simple form, it considers only total matches between records. In order words, comparing two records yield a binary answer that does not account for the degree of difference between them.

However, in real applications, the quality and the correctness of data can be subject to variations due to human errors, cultural differences and lack of standards. As an illustration, consider the different spellings of names in a list of clients or in a product catalog. As another example, for data integration purposes, it might be interesting to detect whether "University of Tsukuba" and "Tsukuba University" refer to the same entity. Applying simple joins to relate records having such subtle differences would not be beneficial. In these scenarios, it is more advantageous to rely on *similarity joins*, a class of joins that acknowledge disparities between records and determine the result based on the degree of similarity between them.

Among the factors pertaining the execution of similarity joins, it is critical to consider the way the data is represented. *Set similarity joins* constitute a variation of similarity joins that works on sets instead of regular records. Although seemingly restrictive at first, set similarity joins provide great usability due to the fact that different types of data can be mapped to sets: market basket data, text and images. Furthermore, by adopting the set representation, it is possible to apply the many well-known similarity metrics used to compare sets (e.g., Jaccard similarity, Dice coefficient and cosine similarity) [55, 75].

Due to their ample applicability, there is a high demand for efficiently processing set similarity joins, especially in the current context of large data. Notably, since performance improving techniques used in simple joins do not consider the similarity factor, a growing body of literature has been proposed to address the specific requirements of similarity joins [2, 5, 14, 41, 69, 77, 87, 119]. The state-of-the-art adopt a filter-verification approach whose objective is to reduce the number of comparisons between pairs of records [2, 14, 41, 119]. This is done by applying filter strategies that first prune dissimilar pairs and generate candidate pairs. Then, it verifies which of the candidate pairs are actually similar.

Other works explore a different processing paradigm, namely parallel processing [77, 87]. Among these, a few researchers have investigated the use of special hardware, accelerators, to speedup similarity joins. In particular, GPUs (graphic processing units) have shown high processing throughput that scales well with highly parallelizable tasks [5, 69].

However, GPUs have a peculiar architecture that differs from that of CPUs in aspects like processing units and memory hierarchy. Furthermore, since the amount of available memory in GPUs is usually modest in comparison to modern CPUs, it is required to carefully design data structures and processing strategies. We propose the utilization of GPUs to accelerate set similarity joins and answer these challenges by leveraging a dimension reduction technique called *MihHash* [9]. By using MinHash to create signatures for the sets, we are able to fit more sets into the GPU's memory, thus improving the overall efficiency of the proposal.

Besides efficiency, another pressing matter affecting set similarity joins relates to the privacy of the data being joined. Due to the growing trend of outsourcing computation and storage to cloud services, there is also an increasing concern related to the privacy of sensitive data (e.g., medical records and proprietary chemical compounds). When joining this kind of data, it is important to do it in a way that there is no disclosure of information other than the final result of the operation. To this end, a promising approach is to use multi-party computation and cryptography to guarantee the secrecy of the data [6, 13, 40, 56, 100, 102].

Fully homomorphic encryption is a cryptography breakthrough [36] that offers high security guarantees while allowing the calculation of similarity between two encrypted records [11, 20, 35, 60, 65, 109, 116, 124]. We propose a two-party set similarity join protocol to securely compute similar pairs of records that come from relations belonging to different parties. Our solution utilizes threshold Tversky index [99] to find similar pairs without leaking their similarity value, thus protecting the computation against regression attacks.

Despite its security and flexibility, fully homomorphic encryption schemes suffer from performance costs that can make their use impractical in certain scenarios. To overcome these performance penalties, we resort to parallelization using multi-core CPUs and to the adaptation of filter techniques to the context of encrypted data. As a result, we achieve important speedups that help making the protocol more feasible in real situations.

We focus on the aforementioned facets of similarity joins (i.e., efficiency and security) and make contributions that include:

- A scheme for set similarity joins that uses the storage efficiency of provided by MinHash [9] and capitalizes on the highly parallel processing power provided by GPUs. The scheme is evaluated in terms of performance and accuracy, and results show high speedups without sacrificing the quality of the results.

- A two-party protocol for secure set similarity joins based on modern fully homomorphic cryptosystems, offering both security and flexibility to perform calculations using encrypted data. To determine whether two sets are similar, we make use of the threshold Tversky index [99], which protects the similarity value itself and prevents its exploitation by regression attacks. In addition, the adaptation of filter techniques and the parallalization using multiple CPU cores assist in reducing the performance costs added by fully homomorphic schemes.

The remainder of this dissertation is organized as follows. Chapter 2 provides more details about similarity joins and about other concepts and technologies used in our solutions. The related work is analysed in Ch. 3, which also positions this dissertation among the existing research. An explanation about our how our efficient scheme for set similarity joins leverages the combination of MinHash and GPUs is presented in Ch. 4. Chapter 5 examines our secure set similarity join protocol, giving security details and offering strategies to improve the performance when computing over encrypted data. Finally, Ch. 6 summarizes this work and discusses future research directions.

Chapter 2

Preliminaries

This chapter presents an overview of the field in which this dissertation is positioned. First, we provide a formal definition of similarity joins and list some of their main applications. Then, we describe the main methods and technologies used in our proposals: Jaccard similarity, GPU, secure multi-party computation and homomorphic encryption.

2.1 Similarity joins

A *similarity join* (Def. 1) is an operator that receives as input two database relations and a similarity threshold, and outputs all pairs of records, one from each relation, whose similarity is greater than the specified threshold. It has become a significant class of database operations due to the diversification of data, and it is used in many applications, such as data cleaning, entity recognition and duplicate elimination [14, 41].

Definition 1. *The similarity join operation takes as input two database relations, R and S, and a similarity threshold θ. It returns pairs of records whose similarity is greater than or equal to θ, according to a given similarity function $Sim(x, y)$:* $\{(r, s)|r \in R, s \in S, Sim(r, s) \geq \theta\}$

Set similarity join [55] is a variation of similarity join that works on sets instead of regular records, and it is an important operation in the family of similarity joins due to its applicability on different data (e.g., market basket data, text and images).

2.1.1 Applications

Due to its high flexibility, similarity joins have been used in a number of different scenarios. Although many applications have some degree of overlap or can be combined, we categorize such applications as follows.

Data Integration

Due to the quantity and the heterogeneity of data processed, in many situations (e.g., enterprise merging, research collaboration, surveillance programs) it is necessary to perform the integration of different data sources [28]. Since such data sources do not follow a standard in terms of identifiers, similarity joins can be used to find common items based on similarity definitions that better suit the situation. In this field, Cohen [17, 18] designed *WHIRL*, a system for similarity joins based on text similarity metrics that infers a rank of possible answers for a given similarity query.

Entity Resolution

Entity resolution, also referred to as record linkage, appertains to finding different database records that correspond to the same entity. Winkler [115] surveyed the use of entity resolution in the context of population census, highlighting challenges like the choice of linking techniques, parameters and confidentiality methodology. Wang et al. [110] introduced an alternative to the common practice of machine-based entity resolution, exploring a hybrid approach that makes use of crowdsourcing to reduce processing costs.

Data Mining

Similarity joins are also used in a number of data mining applications, like social network analysis [103], document clustering [10], recommendation systems [23, 91, 94] and membership checking [12]. Among such works, Chakrabarti et al. [12] proposed a method based on the filter-verification framework to identify input substrings that match with entries of a potentially large dictionary without producing false negatives. Their method uses an in-memory filter structure that first prunes substrings that cannot match with any entry in the dictionary and then verify the remaining substrings.

Data Cleaning

Data cleaning refers to the process of identifying and processing (i.e., correcting or removing) inaccurate or duplicated records in a dataset [1,48]. This is an important step in numerous applications that prepare data for further processing, having great impact on the performance of algorithms and quality of results. One of the most prominent works in this field is the one by Chaudhuri et al. [14], which introduces *prefix filter* (detailed in Sec. 3.1.1), a pruning technique explored in many other works [2,86,111,113,117,119].

Other Applications

Other than the aforementioned applications, a large number of works explored similarity joins in scenarios, ranging from plagiarism detection [49] to finding inflation attacks in advertising networks [76].

2.2 Jaccard Similarity

Among the various set similarity metrics available (e.g., Jaccard similarity index [54], Dice coefficient [27], cosine similarity), one of the most effective is the Jaccard similarity (JS) [51]. We choose to use the Jaccard similarity in this work because of its efficiency and its highly parallelizable calculation method that pairs well with modern multi-core and many-core hardware architectures.

To see how Jaccard similarity is used to calculate the similarity of sets, consider Fig. 2.1. It presents two collections of documents[1], R and S, that contain two documents each: r_0, r_1, and s_0, s_1. In this scenario, the objective of the similarity join is to retrieve pairs of documents, one from each relation, that have a similarity degree greater than a specified threshold θ. One way to represent such documents is to consider them as sets of words (or *tokens*), and then use the Jaccard similarity to determine how similar they are. It is possible to calculate the Jaccard similarity between two sets, A and B, in the following way:

$$JS(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.1}$$

Considering (2.1) and the documents in Fig. 2.1, we obtain the following results:

[1]In a relational database setting, collections correspond to relations, and documents correspond to records. We use these terms interchangeably if there is no ambiguity.

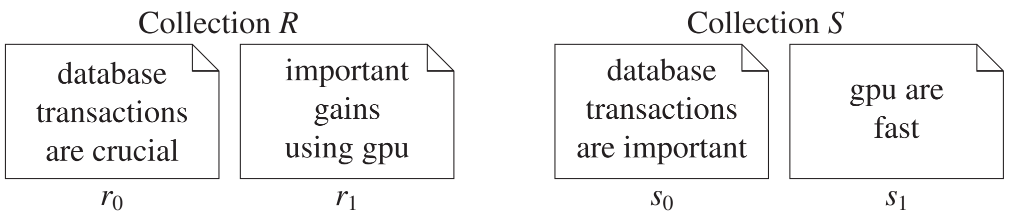

Collection R
database transactions are crucial
important gains using gpu
r_0
r_1
Collection S
database transactions are important
gpu are fast
s_0
s_1

Figure 2.1: Two collections of documents, R and S.

$JS(r_0, s_0) = 3/5 = 0.67$, $JS(r_0, s_1) = 1/6 = 0.17$, $JS(r_1, s_0) = 1/7 = 0.14$ and $JS(r_1, s_1) = 1/6 = 0.17$.

2.3 Graphics Processing Units

Despite being originally designed for games and other graphic applications, the applications of Graphics Processing Units (GPUs) have been extended to general computation due to their high computational power [82]. This section presents the features of this hardware and the challenges encountered when using it.

The properties of a modern GPU can be seen from both a computing and a memory-related perspective (Fig. 2.2). In terms of computational components, the GPU's *scalar processors* (SPs) run the primary processing unit, called *thread*. GPU programs (commonly referred to as *kernels*) run in an SPMD (Single Program Multiple Data) fashion on these lightweight threads. Threads form *blocks*, which are scheduled to run on *streaming multiprocessors* (SMs).

The memory hierarchy of a GPU consists of three main elements: *registers*, *shared memory* and *device memory*. Each thread has access to its own registers (quickly accessible, but small in size) through the register file, but cannot access the registers of other threads. In order to share data among threads in a block, it is possible to use the shared memory, which is also fast, but still small (16KB to 96KB per SM depending on the GPU's capability). Lastly, in order to share data between multiple blocks, the device memory (also called *global memory*) is used. However, it should be noted that the device memory suffers from a long access latency as it resides outside the SMs.

When programming a GPU, one of the greatest challenges is the effective utilization of this hardware's architecture. For example, there are several benefits in exploring the faster memories, as it minimizes the access to the slower device memory and increases the overall performance.

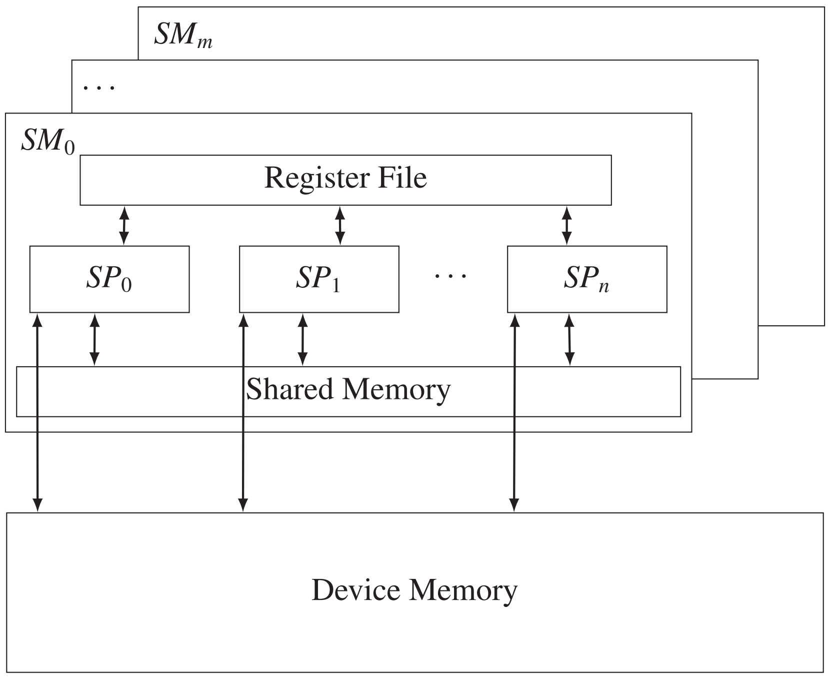


Figure 2.2: Architecture of a modern GPU.

In order to apply a GPU for general processing, it is common to use dedicated libraries that can facilitate such task. Our solution employs NVIDIA's CUDA [80], which provides an extension of the C programming language to define parts of a program to be executed on the GPU.

In terms of algorithms, a number of data-parallel operations, usually called *primitives*, have been ported to be executed on GPUs in order to facilitate programming tasks. He et al. [46, 47] provide details on the design and implementation of many of these primitives.

One primitive particularly useful for our work is *scan* or *prefix-sum* (Def. 2 [45]), which has been target of several works [29, 98, 121]. Figure 2.3 illustrates its basic form, in which the binary operator is addition. It receives as input an array of integers and outputs an array where the value in each position is the sum of the values in previous positions.

Definition 2. *The* scan *(or* prefix-sum*) operation takes a binary associative operator* $\oplus$ *with identity* I*, and an array of* n *elements* $[a_0, a_1, ..., a_{n-1}]$*, and returns the array* $[I, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-2})]$.

As detailed in Sec. 4.3, we use the scan primitive to calculate the positions where each GPU block will write the result of its computation, allowing us to overcome

Input:	2	4	0	1	3
Output:	0	2	6	6	7

Figure 2.3: Example of using the scan primitive.

the lack of incremental memory allocation during the execution of kernels and to avoid write conflicts between blocks. To do so, we adopt the scan implementation provided by the library Thrust [50] due to its high performance and ease of use.

2.4 Secure Multi-party Computation

Multi-party computation pertains to the computation of a given function whose inputs come from the participants of a multi-party protocol. More formally, let us assume that each participant i has an input x_i, and the protocol's objective is to calculate the function $f(x_1, \ldots, x_n)$ using inputs of n participants. Considering secure multi-party computation, one critical requirement is that, at the end of the protocol, the participants should only learn the output of the function. They should not learn about inputs of each other, except what can be derived from the final result and their own original input.

Other than the number of participants, one important concept related to secure multi-party computation is the *adversary model*. Essentially, the adversary model defines the behavior of the involved parties. The term *adversary* can include not only entities attempting to disrupt the protocol, but also the participants themselves, which may be trying to learn more information than initially intended.

It is common to divide adversaries in two categories: honest-but-curious and malicious. Honest-but-curious adversaries, also known as semi-honest, follow the rules of the protocol, but try to learn additional information from it. Such adversaries can also collude to obtain advantages against the non-corrupt parties. On the hand, malicious adversaries are not guaranteed to follow the rules of the protocol. They are able to provide intentionally incorrect input in order to gain advantage during the protocol's execution [100].

One way to help prevent the mentioned leakage of information is to make the data inaccessible to other parties by encrypting it. By doing so, adversaries cannot obtain any new information about the data since they do not have the key for decryption. However, as a drawback, encryption can also limit the usability of the data, since it becomes more difficult to do computations using it. In this context, a number of works, aiming at achieving security and utility, propose methods to computer over encrypted data [6, 13, 40, 56, 102].

Among the findings of such proposals, two main categories of encryption schemes are considered: deterministic and probabilistic. Deterministic cryptosystems [3] yield the same output for the same input (i.e., key and message) given to it. In other words, using a deterministic encryption algorithm E to encrypt two equal messages m_0 and m_1 ($m_0 = m_1$), we would obtain two equal ciphertexts $E(m_0)$ and $E(m_1)$ ($E(m_0) = E(m_1)$). This property can be useful in many applications. For example, consider that one needs to search for a particular record in a database table having deterministically encrypted records. One way to obtain the result is to encrypt the query record using the same cryptosystem and key, and then scan the table to find a matching ciphertext. Nonetheless, deterministic cryptoystems have a major drawback: Although at first the original messages might appear secure, it is possible to learn, for instance, the distribution of a database whose records are deterministically encrypted. This learning itself might be considered an unacceptable information leakage in sensitive applications.

Contrarily, given the same input (m_0, m_1 such that $m_0 = m_1$), probabilistic cryptosystems [39] yield different outputs with high probability ($E(m_0) \neq E(m_1)$). This is a necessary condition to the concept of *semantic security* [38] (or *indistinguishability*), which states that, given a ciphertext $E(m)$, no information can be obtained about the message m with non-negligible probability.

Whilst having great utility, due to the aforementioned properties, deterministic cryptosystems can be target of a number of statistical attacks [15, 59, 61]. On the other hand, probabilistic cryptosystems are generally more secure, but they are also less flexible. Although the decision about what cryptosystem to use depends on the application, probabilistic cryptosystems are usually preferred when processing highly sensitive data.

2.5 Homomorphic Encryption

Other than protect the data, in the case of set similarity joins, it is also important to be able to perform secure similarity computation. To do so, it is necessary to encrypt the data in a way that it can be manipulated. Homomorphic encryption schemes can offer this kind of manipulation while providing the security of probabilistic encryption [32, 83].

For example, consider an encryption algorithm, E, its decryption counterpart, D, and two messages, m_1 and m_2. A cryptosystem having the homomorphic additive property would allow us to sum the corresponding ciphertexts, $E(m_1)$ and $E(m_2)$, and the decryption would yield the sum of the original messages: $D(E(m_1) \oplus E(m_2)) = m_1 + m_2$, where $\oplus$ denotes the homomorphic addition operation.

By being able to do homomorphic additions and multiplications over a ciphertext, one can combine these two basic operations using a logic circuit and do more complex procedures. Although having a system offering both homomorphic addition and homomorphic multiplication was something desired for a long time, it was not until 2009 that the first fully homomorphic encryption scheme was created [36]. Since then, a number of improvements to the original work were proposed [8, 37, 71], and we use one of such improvements, the BGV scheme [8], due to its efficiency when working with large plaintext spaces [19].

Chapter 3

Related Work

This chapter presents the work that relates to this dissertation, describing their approaches and highlighting their contributions. First, we survey studies regarding efficiency and then focus on the privacy-preservation facet of set similarity joins. After that, we establish the position of this dissertation in relation to the existing research.

3.1 Efficient Similarity Joins

Due to the diversification and growth of data seen in recent years, using similarity joins on large datasets has become an usual task. The straightforward approach is to iterate over both relations in a nested-loop fashion and compare the similarity of all the pairs. However, this method has quadratic complexity and does not scale well in the case of large datasets. In order to improve the efficiency of this similarity joins, it is desirable to avoid comparing all pairs, focusing only on the ones that are more likely to be similar.

3.1.1 Filter-verification Framework

A common way to do reduce the computational cost of set similarity joins is to follow the *filter-verification framework*, which first filters pairs of records according to a specified aspect and then calculates similarity values, verifying which pairs are actually similar.

Based on this filter strategy, two of the most important and widely used methods are the length filter [41] and the prefix filter [14].

Length Filter

The length filter [41] considers that two records cannot be similar if their lengths are too different. The original work used string lengths to do such filtering, but the same idea can be used in the case of sets. Although in this dissertation we adapt the conditions in (3.1) to consider the cardinality of sets instead of lengths, we keep the original name of the technique in the remainder of the explanation. Formally, using the Jaccard similarity threshold θ and the cardinalities of sets r and s as $|r|$ and $|s|$, a pair must satisfy the following conditions to pass the length filter:

$$\theta|s| \leq |r| \leq \frac{|s|}{\theta} \tag{3.1}$$

For example, if $\theta = 0.8$ and $|r| = 10$, it is not possible for r to be similar to sets whose cardinality is greater than 12 or lesser than 8. Because of this, we can prune pairs composed by r and such sets.

Prefix Filter

The prefix filter [14] sorts elements in all sets according to a global order and selects the first $p = \lfloor(1 - \theta)|r|\rfloor + 1$ elements as the prefix for set r. If there is no overlap between prefixes of a pair, it is not necessary to calculate the similarity of that pair. For improved efficiency, one can construct an inverted index using the prefixes from one relation, thus reducing the number of prefix comparisons.

To illustrate the prefix filter, consider the steps in Fig. 3.1. This example uses four sets whose elements are words (Fig. 3.1a). In Fig. 3.1b, these sets are sorted using a global order. We used the alphabetical order, but different orders can also be used (e.g., TF-IDF). Then, the prefixes are calculated based on the similarity threshold and the cardinality of the set. For set r_0, the first $p = \lfloor(1 - \theta)|r_0|\rfloor + 1 = \lfloor(1-0.8)5\rfloor+1 = 2$ elements form its prefix (Fig. 3.1c). Finally, instead of comparing prefixes in a nested-loop way, it is possible to create an index using the prefixes from one relation (Fig. 3.1d), and compare the prefixes from the other relation with the index. As result, the pairs (r_1, s_0) and (r_1, s_1) are pruned since the prefixes of sets in those pairs have no overlap.

Section 5.3.6 contains more practical details regarding length and prefix filters, as well as an in-depth discussion about how such filters can improve the performance of set similarity joins over encrypted data.

r_0:	{smart, city, safe, nation, idea}
r_1:	{idea, secure, smart, people, safe}
s_0:	{smart, city, nation, idea}
s_1:	{country, smart, safe, city, people}

(a) Sets from relations R and S.

r_0:	{city, idea, nation, safe, smart}
r_1:	{idea, people, safe, secure, smart}
s_0:	{city, idea, nation, smart}
s_1:	{city, country, people, safe, smart}

(b) Sets' elements sorted according to alphabetical order.

r_0:	{city, idea}
r_1:	{idea, people}
s_0:	{city}
s_1:	{city, country}

(c) Prefixes of each set.

city	$\rightarrow$	s_0, s_1
country	$\rightarrow$	s_1

(d) Inverted index built using prefixes from relation S.

Figure 3.1: Example of prefix filter.

Other Filter Approaches

Other than the length and prefix filters, there are proposals which explore other filters (e.g., suffix filter [118] and position-enhanced length filter [74]), data partitioning methods [26, 72] and other properties of the data, like set relations [112, 114]. For instance, Wang et al. [114] explored relations between sets to reduce the filter phase. They used such relations to skip unnecessary index probing and to incrementally calculate similarity values.

A survey done by Jiang et al. [55] examined a number of string similarity join approaches. The majority of these contributions focus on the elimination of unnecessary comparisons and adopt the filter-verification approach [2, 14, 41, 64, 88, 92, 111, 113, 117, 118]. The evaluated algorithms were divided into categories, depending on the similarity metric they use. In the particular case of Jaccard similarity, AdaptJoin [111] and PPJoin+ [118] yielded the best results. The survey included differences concerning the performance of algorithms based on the size of the dataset and on the length of the joined strings. Jiang et al. [55] also pointed out the necessity for disk-based algorithms to deal with really large datasets that do not fit in memory.

A more recent survey done by Mann et al. [75] performed an extensive experimental evaluation to compare many of the existing similarity join works: All-Pairs [2], PPJoin [119], PPJoin+ [118], GroupJoin [7], MPJoin [88], PEL [74] and AdaptJoin [111]. The survey explored how the filtering techniques used by the different algorithms affect the overall performance. The results showed that All-Pairs [2], PPJoin [119] and GroupJoin [7] have the best performance in a number of different datasets. Mann et al. [75] concluded that, since the verification phase is usually fast, complex techniques that have high pruning power are rarely competitive due to their large overhead during the filter phase.

3.1.2 Parallel Similarity Joins

Instead of solely focusing on the reduction of pairs to verify, other works concentrate on taking advantage of parallel processing to produce more scalable similarity join algorithms [5, 25, 69, 77, 87, 89, 120, 122].

Among these, Vernica et al. [87] proposed a work based on PPJoin+ [118] and used MapReduce [24] to distribute the processing among nodes in CPU clusters, discussing ways to partition the dataset among nodes in order to achieve a good load balancing, as well as methods to control the amount of data kept in-memory during the processing. Metwally et al. [77] claimed speedups of up to 30 times when compared to the work of Vernica et al. [87]. Such improvements were possible due to eliminating the scalability bottleneck caused by prefix filter in a MapReduce setting.

Although the similarity join is a thoroughly discussed topic, works utilizing accelerators, like GPUs, for the processing speedup are not numerous [5, 69]. Lieberman et al. [69] mapped the similarity join operation to a sort-and-search problem and used well-known algorithms and primitives for GPUs to perform these tasks. After applying the bitonic sort algorithm to create a set of space-filling curves from one of the relations, they processed each record of the other relation in parallel, executing searches in the space-filling curves. The similarity between the records was calculated using the Minkowski metric.

Böhm et al. [5] presented two GPU-accelerated nested-loop join (NLJ) algorithms to perform the similarity join operation, and used Euclidean distance to calculate the similarity in both cases. The best of the two methods was the index-supported similarity join, which has a preprocessing phase to create an index structure based on directories. The authors reported that the GPU version of the indexed-supported similarity join achieved an improvement of 4.6 times when compared to its serial CPU version.

3.2 Privacy-preserving Similarity Joins

Despite several studies that attempted to improve similarity joins in terms of performance and accuracy, there are other requirements that create new challenges related to similarity joins. For instance, as the trend of outsourcing data and processing to the cloud grows, the number of concerns about the privacy and security of such data also rise. Even though cloud storage services offer data confidentiality and protection against outside attacks, occurrences of inside threats and leaks indicate that sensitive information should receive extra attention. In this context, privacy-preserving computation has been receiving increasing attention due to its goal of not only protecting the data itself, but also preventing inferences that can ultimately harm the parties involved in a given multi-party protocol [16, 58].

Although focusing in a different application, Vatsalan et al. [107] defined a comprehensive taxonomy for privacy-preserving record linkage that identifies privacy aspects that are also common to similarity joins: number of parties, adversary model and privacy techniques. In particular, the listed privacy techniques include secure hashing encoding [30], secure multi-party computation [123], phonetic encoding [57], anonymization [78], Bloom filters [4] and differential privacy [31].

Among these techniques, we focus on secure multi-party computation due to its security assurances based on strong cryptographic schemes and its applicability to the case of similarity joins [70, 85, 101].

3.2.1 Computing on Encrypted Data

The combination of multi-party computation and encryption as the way to securely perform a given computation is considered by many works [43, 44, 52, 63, 84, 93, 96, 104–107]. Regarding the processing of encrypted data stored in relational databases, most existing proposals make use of different cryptosystems to allow different operations [44, 84, 93, 104].

For instance, Popa et al. designed CryptDB [84], a system that allows SQL queries over encrypted data. The basic flow of execution of CryptDB is as follows: (1) The data owner encrypts their data and send them to the cloud server. (2) Users can then send plaintext queries to a proxy, which parses the query and rewrites it to a secure format by encrypting variables and changing column names. (3) The proxy sends the query to the cloud server, which executes it and returns the encrypted result to the proxy. (4) The proxy decrypts the results and sends them to the users. The implementation is done by user-defined functions incorporated to MySQL [79]. Each record is expanded in a number of new columns that hold the different encryptions of the record. The cryptosystems used include: probabilistic (AES [22] or Blowfish [95] in CBC mode with a random initialization vector), deterministic (Blowfish or AES in CMC mode), order-preserving encryption, homomorphic encryption (Paillier [83]) using ciphertext packing, and word search (SEARCH [102]). The evaluation uses the TPC-C [62] workload and shows a reasonable 21–25% loss in throughput and a storage overhead of about 3.76 times compared to when using plaintext data.

Other systems have been developed based on CryptDB, focusing on different aspects of databases, like analytical processing [104], attribute-based access control [93] and data encrypted using different keys [44]. Nonetheless, none of them consider similarity-related queries.

Despite having a different focus, works related to privacy-preserving similarity calculation can also be found in the field of record linkage [43,52,63,96,105–107]. They explore concepts and techniques that can be also used in the context of set similarity joins, like how to represent and compute the similarity between encrypted data items. For example, Schnell et al. [96] tackled the problem of securely linking identifiers that might contain errors by using Bloom filters [4] generated using cryptographic hash functions (e.g., MD5 [90] and SHA-1 [34]).

3.2.2 Privacy-preserving Similarity Search

Other researchers focus on the similarity search problem, and not joins [11,20,35,60,65,109,116,124].

For instance, Kuzu et al. [60] devised a method to perform similarity search over encrypted data by using locality sensitive hashing (LSH). The similarity searchable encryption scheme used is a non-deterministic method that can generate a trapdoor for a feature of a particular data item. A secure index is created for the data items and it is used to find items that have a specific feature. Although other works also explore the use of trapdoors to perform the similarity computation, they usually have an overhead associated with computation and transferring of trapdoors.

In another study about privacy-preserving similarity search, Wang et al. [108] used Bloom filters [4] to perform privacy-preserving similarity search, exploring locality-sensitive hashing to preserve the similarity between the original records.

3.2.3 Privacy-preserving Similarity Joins

Up to date, there is not a large number of proposals that tackle the problem of privacy-preserving similarity joins [97, 125]. Although focusing on the privacy-preserving record linkage problem, Sehili et al. [97] adapted PPJoin [119] to data encoded in Bloom filters and to the GPU architecture, achieving speedups of 20 times compared to sequential implementations. Since their data representation was based on the encryption strategy by Schnell et al. [96] (i.e., Bloom filters constructed using deterministic functions), their scheme was also susceptible to common statistical attacks [15].

Yuan et al. [125] explored the combination of locality-sensitive hashing (LSH) [53] and searchable symmetric encryption (SSE) [21]. First, one of the parties, the data owner, creates a LSH-based index. Then, the client generates tokens to probe the index, and count the number of LSH collisions that happen for a given token. If this number is greater than a given threshold, the pair is considered similar. They proposed different schemes that try to achieve a balance between security and performance, using a client cache to avoid sending repeated tokens and hiding the distribution of the relations. Despite having good performance, the security of the system relies on deterministic pseudo-random functions and might not be secure enough for highly sensitive applications.

3.3 Position of this Dissertation

This dissertation addresses two important aspects of set similarity joins: efficiency and security. The main objective is to enable fast processing of large datasets and to protect the privacy of sensitive data during this processing. To this end, we propose solutions that answer these challenges by harnessing recent technologies, namely general-purpose processing on GPUs and fully homomorphic encryption.

Concerning efficiency, our first solution exploits the massive parallel processing power provided by GPUs and achieve high speedups in the processing of set similarity joins. The main characteristic that discerns our work from other similarity join schemes that use GPUs is the effective use of MinHash [9] to overcome challenges inherent to the use of GPUs, especially the memory limitations.

Regarding security, we design a novel two-party set similarity join protocol that is based on the strong security guarantees of modern fully homomorphic cryptography schemes. By leveraging the threshold Tversky index [99], the solution outputs similar pairs without leaking the similarity value between them. In addition, we present the adaptation of established filter strategies to a privacy-preserving context, thus overcoming performance penalties of homomorphic cryptosystems. To the best of our knowledge, this is the first protocols to apply the considerably recent fully homomorphic cryptosystems to execute similarity joins.

We expect our contributions to aid in the processing of large datasets and to add to the growing area of secure processing in untrusted environments.

Chapter 4

GPU Acceleration of Set Similarity Joins

In this chapter, we propose a scheme for efficient set similarity joins. Our solution takes advantage of the massive parallel processing offered by GPUs. Additionally, we employ MinHash to estimate the similarity between two sets in terms of Jaccard similarity. By exploiting the high parallelism of GPUs and the space efficiency provided by MinHash, we can achieve high performance without sacrificing accuracy. Experimental results show that our proposed method is more than two orders of magnitude faster than the serial version of CPU implementation, and 25 times faster than the parallel version of CPU implementation, while generating highly precise query results.

4.1 Introduction

One of the major drawbacks of a set similarity join is that it is a computationally demanding task, especially considering the rapidly increasing sizes of datasets seen nowadays. For this reason, many researchers have proposed different set similarity join processing schemes [92, 111, 118]. Among them, it has been shown that parallel computation is a cost-effective option to tackle this problem [77, 87], notably with the use of Graphics Processing Units (GPUs), which have been gaining much attention due to their performance in general processing [82].

Nonetheless, there are numerous technical challenges when performing set similarity join using GPUs. First, how to deal with large datasets using GPU's memory, which is limited up to a few GBs in size. Second, how to make the best use of the high parallelism of GPUs in different stages of the processing (e.g., similarity computation and the join itself). Third, how to take advantage of the different types of memories on GPUs, such as device memory and shared memory, in order to maximize the performance.

We propose a new scheme of set similarity join on GPUs. To address the aforementioned technical challenges, we employ MinHash [9] to estimate the similarity between two sets in terms of their Jaccard similarity. MinHash is known to be a space-efficient algorithm to estimate the Jaccard similarity, while making it possible to maintain a good trade-off between accuracy and computation time. Moreover, we carefully design data structures and memory access patterns to exploit the GPU's massive parallelism and achieve high speedups. Experimental results show that our proposed method is more than two orders of magnitude faster than the serial version of CPU implementation, and 25 times faster than the parallel version of CPU implementation. In both cases, we assure the quality of the results by maximizing precision and recall values. We expect that such contributions can be effectively applied to process large datasets in real-world applications.

4.2 Estimation of the Jaccard Similarity

Although conceptually simple, the computation of Jaccard similarity (explained in Sec. 2.2) requires a number of pairwise comparisons among the elements from different sets to identify common elements, which incurs a long execution time, particularly when the sets being compared are large. In addition, it is necessary to store the whole sets in memory, which can require prohibitive storage [68].

To address the aforementioned problems, Broder et al. proposed a technique called MinHash (Min-wise Hashing) [9]. Its main idea is to create signatures for each set based on its elements and then compare the signatures to estimate their Jaccard similarity. If two sets have many coinciding signature parts, they have a high degree of similarity with high probability. In this way, it is possible to estimate the Jaccard similarity without conducting costly scans over all elements. In addition one only needs to store the signatures instead of all the elements of the sets, which greatly contributes to reduce the required storage.

After its introduction, Li et al. suggested a series of improvements for the MinHash technique related to memory use and computation performance [66–68]. Our proposal is based on the latest of those improvements, namely, *One Permutation Hashing* [67].

In order to estimate the similarity of the documents in Figure 2.1 using One

	r_0	r_1	s_0	s_1
database	1	0	1	0
transactions	1	0	1	0
are	1	0	1	1
crucial	1	0	0	0
important	0	1	1	0
gains	0	1	0	0
using	0	1	0	0
gpu	0	1	0	1
fast	0	0	0	1

(a) Before row permutation

	r_0	r_1	s_0	s_1	
fast	0	0	0	1	bin b_0
important	0	1	1	0	
gains	0	1	0	0	
database	1	0	1	0	bin b_1
are	1	0	1	1	
crucial	1	0	0	0	
gpu	0	1	0	1	bin b_2
using	0	1	0	0	
transactions	1	0	1	0	

(b) After row permutation

Figure 4.1: Characteristic matrices constructed based on the documents from Figure 2.1, before and after a permutation of rows.

Permutation Hashing, first we change their representation to a data structure called *characteristic matrix* (Figure 4.1a), which assigns the value *1* when a token represented by a row belongs to a document represented by a column, and *0* when it does not.

After that, in order to obtain an unbiased similarity estimation, a random permutation of rows is applied to the characteristic matrix, followed by a division of the rows into partitions (henceforth called *bins*) of approximate size (Figure 4.1b). However, since the actual permutation of rows in a large matrix constitutes an expensive operation, in practice, MinHash uses hash functions to simulate such permutation.

Compared to the original MinHash approach [9], One Permutation Hashing presents a more efficient strategy for computation and storage, since it computes only one permutation instead of a few hundreds. For example, considering a dataset with D (e.g., 10^9) features, each permutation emulated by a hash function would require a an array of D positions. Considering a large number k (e.g., $k = 500$) of hash functions, a total of $D \times k$ positions would be needed for the scheme, thus making the storage requirements impractical for many large-scale applications [67].

For each bin, each document has a value that will compose its signature. This value is the index of the row containing the first 1 (scanning the matrix in a top-down fashion) in the column representing the document. For example, the signature for the document s_0 is 1, 3 and 8. It can happen that a bin for a given document does not have any value (e.g., the first bin of set r_0), and this is also taken into consideration during the similarity estimation. Figure 4.2 shows a data structure called *signature*

	b_0	b_1	b_2
R_0	*	3	8
R_1	1	*	6
S_0	1	3	8
S_1	0	4	6

Figure 4.2: Signature matrix, with columns corresponding to the bins composing the signatures of documents, and rows corresponding to the documents themselves. The symbol * denotes an empty bin.

matrix, which contains the signatures obtained for all the documents.

Finally, the similarity between any two documents is estimated by Eq. 4.1, where N_{mat} is the number of matching bins between the signatures of the two documents, b represents the total number of bins composing the signatures, and N_{emp} refers to the number of matching empty bins.

$$Sim(X, Y) = \frac{N_{mat}}{(b - N_{emp})} \tag{4.1}$$

The estimated similarities for the given example are $Sim(r_0, s_0) = 2/3 = 0.67$, $Sim(r_0, s_1) = 0/3 = 0$, $Sim(r_1, s_0) = 1/3 = 0.33$ and $Sim(r_1, s_1) = 1/3 = 0.33$. Even in this simple example, the estimated values can are close to the real Jaccard similarities previously calculated (i.e., 0.67, 0.17, 0.14 and 0.17). In practical terms, using more bins yields a more accurate estimation, but it also increases the size of the signature matrix.

Let us observe an important characteristic of MinHash. Since the signatures are independent of each other, it presents a good opportunity for parallelization. Indeed, the combination of MinHash and parallel processing using GPUs has been considered by Li et al. [68], as they showed a reduction of the processing time by more than an order of magnitude in online learning applications. While their focus was the MinHash itself, here we use it as a tool in the similarity join processing.

4.3 Efficient Similarity Joins

In the following discussion, we consider the sets to be text documents stored on disk, but the solution can be readily adapted to other types of data. We also assume that techniques to prepare the data for processing (e.g., stop-word removal

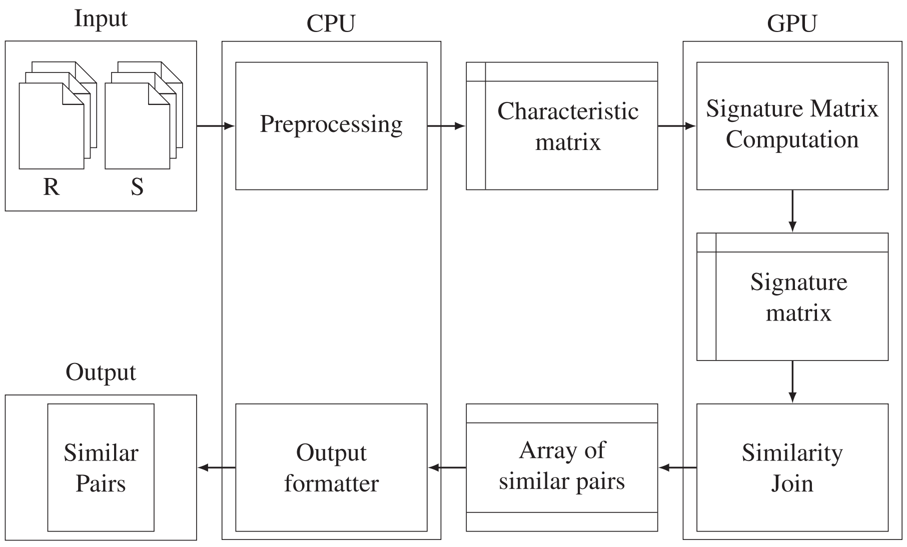


Figure 4.3: System's workflow.

and stemming) are out of our scope, and should take place before the similarity join processing on GPU.

Figure 4.3 shows the workflow of the proposed scheme. First, the system receives two collections of documents representing relations R and S. After that, it executes the three main steps of our solution: preprocessing, signature matrix computation and similarity join. Finally, the result can be presented to the user after being properly formatted.

4.3.1 Preprocessing

In the preprocessing step, we construct a compact representation of the characteristic matrix, since the original one is usually highly sparse. By doing so, the amount of data to be transferred to the GPU is greatly reduced (more than 95% for the datasets used in the experimental evaluation in Section 4.4).

This representation is based on the Compressed Row Storage (CRS) format [42], which uses three arrays: *var*, which stores the values of the nonzero elements of the matrix; *col_ind*, that holds the column indexes of the elements in the *var* array; and *row_ptr*, which points to the locations in the *var* array that start a row in the matrix.

Considering that the nonzero elements of the characteristic matrix have the same value, *1*, there is only need to store their positions. Figure 4.4 shows such representation for the characteristic matrix of the previous example (Fig. 4.1a). The array

	r_0	r_1	s_0	s_1											
doc_start	0	4	8	12	15										
doc_tok	0	1	2	3	4	5	6	7	0	1	2	4	2	7	8

Figure 4.4: Compact representation of the characteristic matrix.

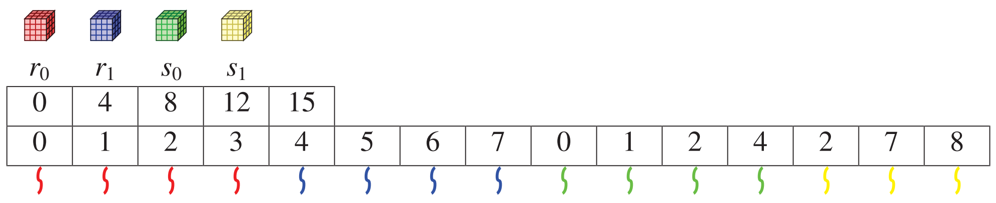


Figure 4.5: Computation of the signature matrix based on the characteristic matrix. Each GPU block is responsible for one document, and each thread is assigned to one token.

doc_start holds the positions in the array *doc_tok* where the documents start, and the array *doc_tok* shows what tokens belong to each document.

After its construction, the characteristic matrix is sent to the GPU, and we assume it fits completely in the device memory. If the characteristic matrix does not fit in the device memory, it is possible to process it in a nested-block fashion. In addition, the overlapping of data transfer and processing can also contribute to handle larger datasets.

4.3.2 Signature Matrix Computation

Once the characteristic matrix is in the GPU's device memory, the next step is to construct the signature matrix. Algorithm 1 shows how we parallelize the MinHash technique, and Fig 4.5 illustrates this processing. In practical terms, one block is responsible for computing the signature of one document at a time. Each thread in the block (1) accesses the device memory, (2) retrieves the position of one token of the document, (3) applies a hash function to it to simulate the row permutation, (4) calculates which bin the token will fit into, and (5) updates that bin. If more than one value is assigned to the same bin, the algorithm keeps the minimum value (hence the name MinHash).

During its computation, the signature for the document is stored in the shared memory, which supports fast communication between the threads of a block. This is advantageous in two aspects: (1) It allows fast updates of values when constructing the signature matrix, and (2) since different threads can access sequential memory

Algorithm 1: Parallel MinHash.

```
input : characteristic matrix $CM_{t \times d}$ ($t$ tokens, $d$ documents), number of bins $b$
output: signature matrix $SM_{d \times b}$ ($d$ documents, $b$ bins)
1  binSize ← $\lfloor t/b \rfloor$;
2  for $i \leftarrow 0$ to $d$ in parallel do // executed by blocks
3      for $j \leftarrow 0$ to $t$ in parallel do // executed by threads
4          if $CM_{j,i} = 1$ then
5              $h \leftarrow hash(CM_{j,i})$;
6              $binIdx \leftarrow \lfloor h/binSize \rfloor$;
7              $SM_{i,binIdx} \leftarrow min(SM_{i,binIdx}, h)$;
8          end
9      end
10 end
```

positions, it favors coalesced access to the device memory when the signature computation ends. Accessing the device memory in a coalesced manner means that a number of threads will access consecutive memory locations, and such accesses can be grouped into a single transaction. This makes the transfer of data from and to the device memory much faster.

The complete signature matrix is laid out in the device memory as a single array of integers. Since the number of bins per signature is known, it is possible to perform direct access to the signature of any given document.

After the signature matrix is constructed, it is kept in the GPU's memory to be used in the next step: the join itself. This also minimizes data transfers between CPU and GPU.

4.3.3 Join

The next step is the similarity join, and it utilizes the results obtained in the previous phase, i.e., the signatures generated using MinHash. To address the similarity join problem, we choose to parallelize the nested-loop join (NLJ) algorithm. The nested-loop join algorithm iterates through the two relations being joined and check whether the pairs of records, one from each relation, comply with a given predicate. For the similarity join case, this predicate is that the records of the pairs must have a degree of similarity greater than a given threshold.

Algorithm 2 outlines our parallelization of the NLJ for GPUs. Initially, each

Algorithm 2: Parallel nested-loop join.

```
input  : signature matrix $SM_{d \times b}$ ($d$ documents, $b$ bins), similarity threshold $\theta$
output: pairs of sets whose similarity is greater than $\theta$
1  foreach $r \in R$ in parallel do // executed by blocks
2      $r\_signature \leftarrow SM_r$;// read the row corresponding to the
         signature of $r$ and store it in the shared memory
3      foreach $s \in S$ in parallel do // executed by threads
4          $coinciding\_minhashes \leftarrow 0$;
5          $empty\_bins \leftarrow 0$;
6          for $i \leftarrow 0$ to $b$ do
7              if $r\_signature_i = SM_{s,i}$ then
8                  if $r\_signature_i$ is empty then
9                      $empty\_bins \leftarrow empty\_bins + 1$;
10                 else
11                     $coinciding\_minhashes \leftarrow coinciding\_minhashes + 1$;
12                 end
13             end
14         end
15         $pair\_similarity \leftarrow coinciding\_minhashes/(b - empty\_bins)$;
16         if $pair\_similarity \geq \theta$ then
17             output($r$, $s$);
18         end
19     end
20 end
```

block reads the signature of a document from collection R and copies it to the shared memory (line 2, Fig. 4.6a). Then, threads compare the value of each bin of that signature to the corresponding signature bin of a document from collection S (lines 3–7), checking whether they match and whether the bin is empty (lines 8–12). The access to the data in the device memory is done in a coalesced manner, as illustrated by Fig. 4.6b. Finally, using Equation 4.1, if the comparison yields a similarity greater than the given threshold (line 15–16), that pair of documents belongs to the final result (line 17).

As highlighted by He et al. [47], outputting the result from a join performed in the GPU raises two main problems. First, since the size of the output is initially unknown, it is also not possible to know how much memory should be allocated on the GPU to hold the result. In addition, there may be conflicts between blocks when

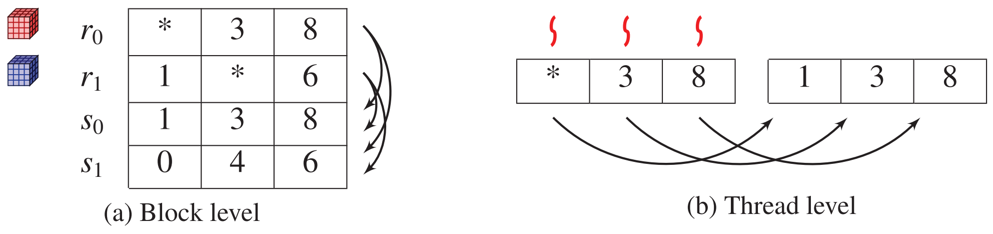


Figure 4.6: Parallelization of NLJ.

writing on the same locations of the device memory. For this reason, He et al. [47] proposed a join scheme for result output that allows parallel writing, which we also adopt in our implementation.

Their join scheme performs the join in three phases:

1. The join is run once, and the blocks count the number of similar pairs found in their portion of the execution, writing this amount in an array stored in the device memory. There is no write conflict in this phase, since each block writes in a different position of the array.

2. Using the scan primitive, it is possible to know the correct size of memory that should be allocated for the results, as well as where the threads of each block should start writing the similar pairs they found.

3. The similarity join is run once again, outputting the similar pairs to the proper positions in the allocated space.

This process is illustrated in Fig. 20. First, four blocks write the size of their results in the first array. Then, the scan primitive gives the starting positions where each block should write. Finally, each block writes its results in the last array.

After that, depending on the application, the pairs can be transferred back to the CPU and output to the user or kept in the GPU for further processing by other algorithms.

4.4 Experimental Evaluation

In this section, we present the experiments performed to evaluate our proposal. First, we introduce the used datasets and the environment on which the experiments

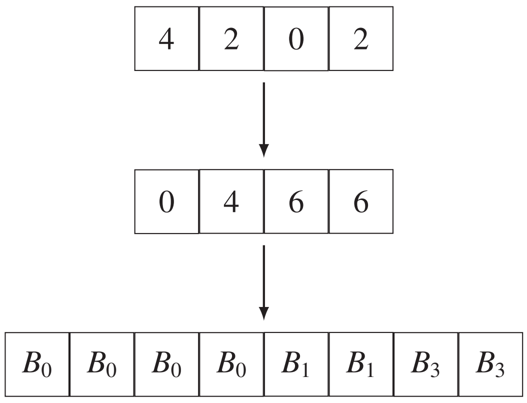


Figure 4.7: Example of the three-phase join scheme created by He et al. [47].

Table 4.1: Characteristics of datasets.

Dataset	Number of Sets	Average Number of Elements per Set
IMAGES	68,040	32
ABSTRACTS	233,445	165
TRANSACTIONS	1,692,082	177

were conducted. After that, we show the results related to performance and accuracy. Finally, we present other experiments related to parameter tuning and to the behavior when varying characteristics of relations.

4.4.1 Setup

To demonstrate the range of applicability of our work, we chose datasets from three distinct domains (Table 4.1). The *IMAGES* dataset, made available at the UCI Machine Learning Repository[1], consists of image features extracted from the Corel image collection. The *ABSTRACTS* dataset, composed by abstracts of publications from MEDLINE, were obtained from TREC-9 Filtering Track Collections[2]. Finally, *TRANSACTIONS* is a transactional dataset available through the FIMI repository[3].

From the original datasets, we randomly chose sets in order to create collections R and S, whose sizes vary from 1,024 to 524,288 sets. Unless stated otherwise, all

[1]http://archive.ics.uci.edu/ml/datasets/
[2]http://trec.nist.gov/data/t9_filtering.html
[3]http://fimi.ua.ac.be/data/

the experiments were performed using the same parameters and settings, including the contents of the relations. The similarity threshold used was 0.8 and the number of bins composing the documents' signatures was 32.

The CPU used in our experiments was an Intel Xeon E5-1650 (6 cores, 12 threads) with 32GB of memory. The GPU was an NVIDIA Tesla K20Xm (2,688 scalar processors) with 6GB of memory. Regarding the compilers, GCC 4.4.7 (with the flag -O3) was used for the part of the code to run on the CPU, and NVCC 6.5 (with the flags -O3 and -use_fast_math) compiled the code for the GPU. For the parallelization of the CPU version, we used OpenMP 4.0 [81].

4.4.2 Performance Comparison

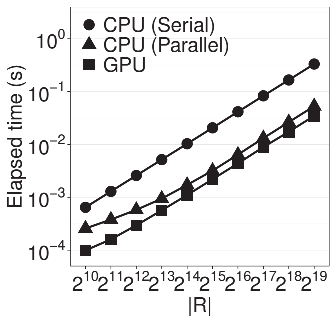
Figure 4.8, Fig. 4.9 and Fig. 4.10 present the execution time of our approach for the three implementations (GPU, CPU Parallel and CPU Serial) using the three datasets.

Let us first consider the MinHash part, i.e., the time taken for the construction of the signature matrix. It can be seen from the results (Fig 4.8a, Fig. 4.8b and Fig. 4.8c) that the GPU version of MinHash is more than 20 times faster than the serial implementation on CPU, and more than 3 times faster than the parallel implementation on CPU. These findings reinforce the idea that MinHash is indeed suitable for parallel processing.

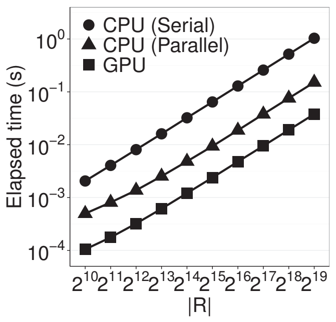
For the join part (Fig. 4.9a, Fig. 4.9b and Fig. 4.9c), the speedups are even higher. The GPU implementation is more than 150 times faster than the CPU Serial implementation, and almost 25 times faster than the CPU Parallel implementation. The speedups of more than two orders of magnitude demonstrate that the NLJ algorithm can benefit from the massive parallelism provided by GPU.

Measurements of the total time of execution (Fig. 4.10a, Fig. 4.10b and Fig. 4.10c) show that the GPU implementation achieves speedups of approximately 120 times when compared to the CPU Serial implementation, and approximately 20 times when compared to the CPU Parallel implementation.

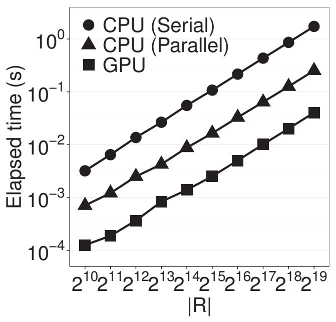
The analysis of performance details provides some insights into why the overall speedup is lower than the join speedup. Table 4.2, Tab. 4.3 and Tab. 4.4 present the breakdown of the execution time for each of the datasets used. Especially for larger collections, the join step is the most time consuming part for both CPU implementations. However, for the GPU implementation, reading from data disk becomes the bottleneck, as it is done in a sequential manner by the CPU. Therefore, since the overall measured time includes reading data from disk, the speedup achieved is less than the one for the join step alone. A mentioned in Sec. 4.3.1, overlapping data



(a) IMAGES

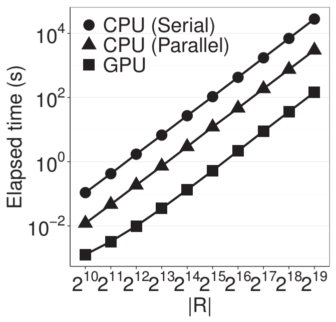


(b) ABSTRACTS

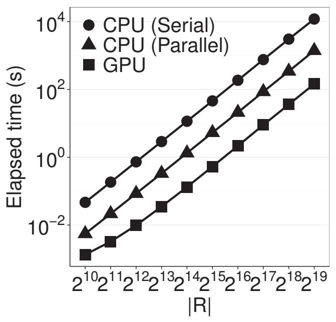


(c) TRANSACTIONS

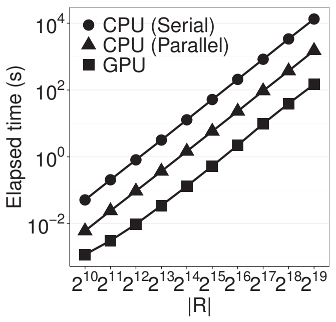
Figure 4.8: MinHash performance comparison ($|R| = |S|$).



(a) IMAGES

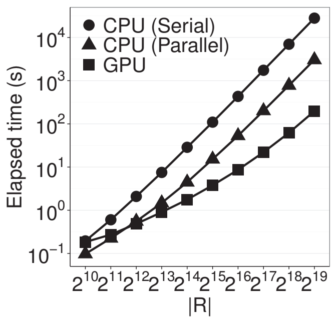


(b) ABSTRACTS

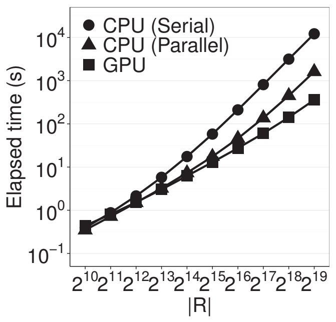


(c) TRANSACTIONS

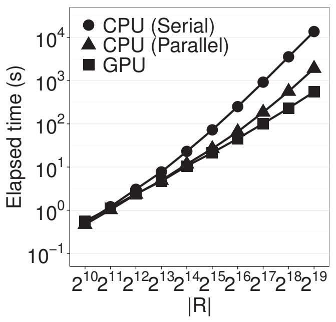
Figure 4.9: Join performance comparison ($|R| = |S|$).



(a) IMAGES



(b) ABSTRACTS



(c) TRANSACTIONS

Figure 4.10: Overall performance comparison ($|R| = |S|$).

reading and processing can be help lessen the impact of such a bottleneck.

It can also be noted that the compact data structures used in the solution contribute directly for the short data transfer time between CPU and GPU. In the case of the CPU implementations, this transfer time does not apply, since the data stays on the CPU throughout the whole execution.

4.4.3 Accuracy Evaluation

Since our scheme uses the MinHash technique to estimate the similarity between sets, it is also important to confirm how accurate are the results obtained from it. We evaluated the accuracy of the proposal in terms of precision and recall. Precision relates to the fraction of really similar pairs among all the pairs retrieved by the algorithm, and recall refers to the fraction of really similar pairs that were correctly retrieved.

Table 4.5 presents the measurements of experiments in which we varied the number of bins composing the signatures of the documents, showing the impact of the number of bins on the number of similar pairs found, as well as on the performance. As the number of bins increases, the number of similar pairs found nears the number of really similar pairs, thus increasing the values of precision and recall. On the other hand, increasing the number of bins also incurs a longer execution time. Therefore, it is important to achieve a balance between accuracy and execution time. For the used datasets, using 32 bins offered a good trade-off, yielding the lowest execution time without false positive or false negative results.

4.4.4 Other Experiments

We also conducted experiments varying other parameters of the implementation and characteristics of the datasets. Fig. 4.11 shows that, in the GPU implementation, varying the number of threads per block has little impact on the performance.

Figure 4.12 reveals that all three implementations are not greatly affected by varying the similarity threshold. In other words, although the number of similar pairs found changes, the GPU implementation is consistently faster than the other two.

Table 4.6 shows the impact of the similarity threshold on precision and recall levels. When the threshold is low, many pairs with a low degree of similarity are also part of the result, thus increasing the number of false positives. This situation is illustrated by the low precision value when the similarity threshold is 0.2.

Table 4.2: Breakdown of the execution time in seconds when joining collections of the same size (IMAGES dataset, $|R| = |S| = 524,288$).

	GPU	CPU (Parallel)	CPU (Serial)
Read from disk	47.7	47.2	47.4
Preprocessing	2.9	2.9	2.9
MinHash	0.034	0.053	0.332
Join	145	2,988	27,964
Data transfer	0.53	0	0
Total	197	3,040	28,016

Table 4.3: Breakdown of the execution time in seconds when joining collections of the same size (ABSTRACTS dataset, $|R| = |S| = 524,288$).

	GPU	CPU (Parallel)	CPU (Serial)
Read from disk	201.5	200.5	198.4
Preprocessing	9.3	9.4	9.1
MinHash	0.037	0.151	1.033
Join	145	1,403	11,955
Data transfer	0.09	0	0
Total	359	1,615	12,167

Table 4.4: Breakdown of the execution time in seconds when joining collections of the same size (TRANSACTIONS dataset, $|R| = |S| = 524,288$).

	GPU	CPU (Parallel)	CPU (Serial)
Read from disk	379.8	378.4	376.2
Preprocessing	15.9	16.1	15.6
MinHash	0.040	0.250	1.728
Join	147	1,513	13,323
Data transfer	0.21	0	0
Total	549	1,914	13,723

Table 4.5: Impact of varying number of bins on precision, recall and execution time (ABSTRACTS dataset, $|R| = |S| = 65,536$).

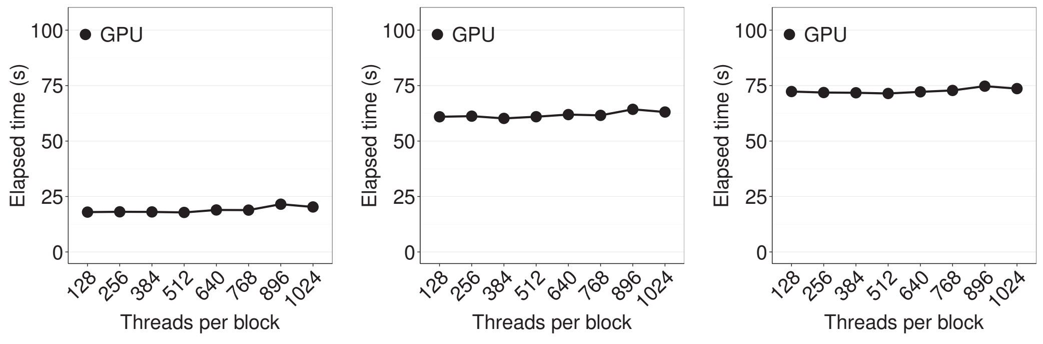
Number of Bins	Precision	Recall	Execution Time (s)
1	0.0000	0.9999	25.3
2	0.0275	0.9999	25.4
4	0.9733	0.9999	25.6
8	0.9994	0.9999	25.7
16	0.9998	1.0000	26.1
32	1.0000	1.0000	27.4
64	1.0000	1.0000	29.6
128	1.0000	1.0000	34.4
256	1.0000	1.0000	45.8
384	1.0000	1.0000	77.6
512	1.0000	1.0000	133.6
640	1.0000	1.0000	161.5

Additionally, we constructed different collections of sets by varying the number of matching sets between them, i.e., the join selectivity. Figure 4.13 indicates that varying the selectivity does not impact the join performance.

Finally, we also investigated the performance when joining collections of different sizes. Figure 4.14 shows the overall results when we vary the size of the collection processed the outer loop. Similar results were found when varying the size of the collection processed by the inner loop.

Table 4.6: Precision and recall varying similarity threshold (GPU implementation, ABSTRACTS dataset, $|R| = |S| = 8,192$).

Similarity Threshold	Precision	Recall
0.2	0.08	0.89
0.4	0.99	0.99
0.6	1.00	1.00
0.8	1.00	1.00
1.0	1.00	1.00



(a) IMAGES (b) ABSTRACTS (c) TRANSACTIONS

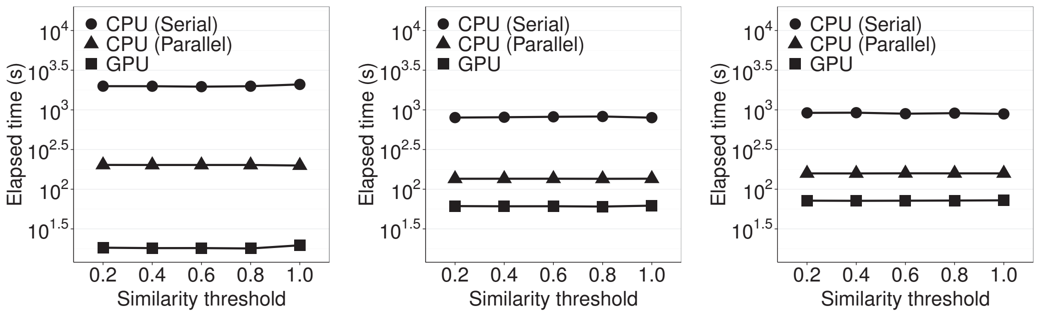
Figure 4.11: Execution time varying the number of threads per GPU block ($|R| = |S| = 131,072$).

4.5 Summary

In this chapters, we have proposed a GPU-accelerated similarity join scheme that uses MinHash in its similarity calculation step. Experiments have shown a speedup of more than two orders of magnitude when compared to the sequential version of the algorithm. Moreover, the high levels of precision and recall confirmed the accuracy of our scheme.

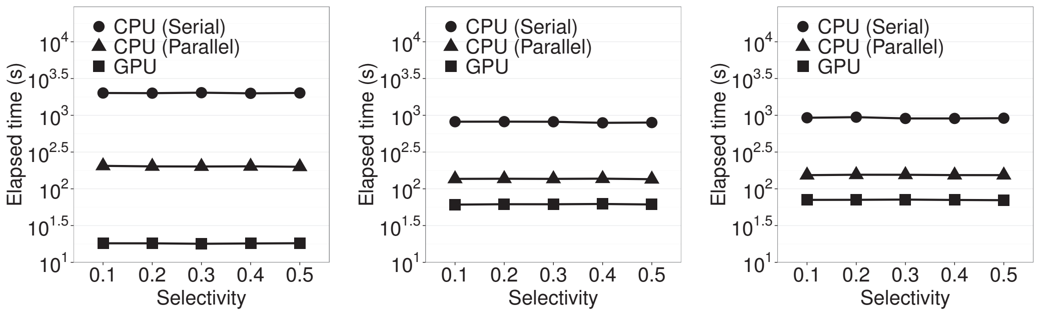
The strongest point of GPUs is their superior throughput when compared to CPUs. However, they require special implementation techniques to minimize memory access and data transfer. For this purpose, using MinHash to estimate the similarity of sets is particularly beneficial, since it enables a parallelizable way to represent the sets in a compact manner, thus saving storage and reducing data transfer. Furthermore, our implementation explored the faster GPU memories (i.e., registers and shared memory) to diminish effects of memory stalls. This solution can aid in the task of processing large datasets in a cost-effective way without ignoring the quality of the results.

The main characteristic that discerns our work from the other similarity join schemes for GPUs is the effective use of MinHash to overcome challenges inherent to the use of GPUs for general-purpose computation, as emphasized in Sec 4.2. Furthermore, to the best of our knowledge, our solution is the first one to couple Jaccard similarity and GPUs to tackle the similarity join problem.



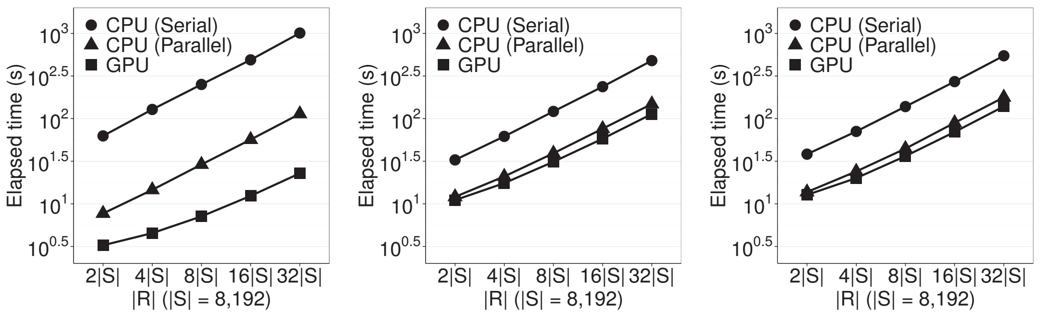
(a) IMAGES (b) ABSTRACTS (c) TRANSACTIONS

Figure 4.12: Execution time varying the similarity threshold ($|R| = |S| = 131,072$).



(a) IMAGES (b) ABSTRACTS (c) TRANSACTIONS

Figure 4.13: Execution time varying the join selectivity ($|R| = |S| = 131,072$).



(a) IMAGES (b) ABSTRACTS (c) TRANSACTIONS

Figure 4.14: Overall performance comparisons varying the size of the outer relation.

Chapter 5

Privacy-Preserving Set Similarity Joins

Despite the numerous contributions regarding efficiency and performance of similarity joins, the privacy of the data being joined also becomes an important aspect to consider, as leaking sensitive information can result in grave consequences for individuals, enterprises and governmental organizations. In this chapter, we propose a protocol for secure execution of similarity joins that is based on fully homomorphic cryptosystems, which are resistant to a number of attacks and provide flexibility to calculate the similarity between encrypted records. We also consider the adaptation of filter techniques to improve the efficiency of the protocol by reducing the number of record pairs that are compared. In addition, we exploit modern hardware to parallelize the solution and evaluate the performance of the proposal using real datasets.

5.1 Introduction

Although the main concerns related to similarity joins are usually focused on performance, in some situations the security and the privacy of the data is paramount.

For instance, consider a scenario in which the administrative body of a particular city wants to make a study regarding the health of the citizens. Analyzing data to find common items from different sources may reveal patterns that can be used to improve the health system of a particular region. To do so, it may be necessary to gather data from different institutions, like hospitals, clinics and laboratories. Some records might have subtle differences in the way they are stored, even though they might refer to the same entity. This is the case when a patient's name has a typo in one of the databases or the middle name is abbreviated. Joining these records based on their similarity provides a way to take such differences into account and improve the quality of the results.

Despite the efforts to enhance similarity joins in terms of both accuracy and performance [111, 119], the privacy of the data being joined also becomes an important aspect to consider, as leaking sensitive information can result in grave consequences for individuals and enterprises alike. In the aforementioned example, medical records have information that, if leaked, could result in more expensive insurance plans or discrimination against a particular individual. The possibility of such harmful consequences raises security requirements for the use of sensitive data. Regarding similarity joins, one of these requirement is that the result of the join can be disclosed (i.e., the common records), but not the records that do not belong to the result.

Encrypting data in a way that allows processing over ciphertexts has emerged as a solution to this problem, as it considers both utility (i.e., the possibility of similarity calculation) and security. Such a solution has been adopted by different protocols to address applications like secure similarity search and record linkage [60, 63, 65, 105], but it has not been much explored in the context of similarity joins. Furthermore, the few existing works using encryption base their security on deterministic cryptographic functions [97], which are vulnerable to statistical attacks [15].

To address these limitations, we propose a two-party protocol that securely executes set similarity joins over encrypted data. Parties are able to identify common records in their databases without disclosing the rest of their records. The encryption is done using a fully homomorphic cryptosystem, which provides semantic security (thus avoiding statistical attacks) and enough flexibility to manipulate ciphertexts. This allows us to use the additive and multiplicative properties of such schemes to calculate the similarity between pairs of encrypted records.

The similarity computation uses the threshold Tversky index [99], which is a similarity measure that generalizes the Dice coefficient and the Jaccard similarity index. The advantages of using the threshold Tversky index are two-fold. First, it allows the calculation of similarity between encrypted records by using simple operations, namely addition and multiplication of ciphertexts. Second, it does not leak the similarity value itself, thus avoiding regression attacks.

Although useful, current fully homomorphic cryptosystems present limitations in terms of performance. To address this drawback, we propose secure adaptations of existing filter techniques (i.e., length filter and prefix filter [14, 41]) that can prune pairs of highly dissimilar records, thus reducing the total number of comparisons done and the overall execution time. Furthermore, we explore parallelization opportunities throughout the protocol in order to diminish the overhead created by the use of fully homomorphic encryption.

Our contributions include:

- A two-party protocol for secure execution of similarity joins. The protocol uses a semantically secure homomorphic cryptosystem that provides strong security while offering ways to calculate similarity over encrypted data.
- Strategies to adapt consolidated filtering techniques to the context of encrypted data. These techniques aim at reducing the number of pairs to be compared by the join step, thus improving the overall efficiency of the approach.
- Parallelization of the protocol to exploit modern hardware architecture and improve the performance.
- Experimental evaluation of the proposal using real datasets, focusing on the performance comparison between different optimizations.

In the remainder of this chapter, we utilize the notations summarized in Tab. 5.1.

5.2 Secure Similarity Calculation

Due to the limited number of operations offered by fully homomorphic cryptosystems, the similarity computation should be done using only additions and multiplications. This can be achieved by the threshold Tversky index [99].

The Tversky index ($TI_{\alpha,\beta}$), given by (5.1), can be seen as a generalization of the more well-known Jaccard similarity ($\alpha = \beta = 1$) and Dice coefficient ($\alpha = \beta = 1/2$). Considering a similarity threshold θ, if the Tversky index between two vectors is equal to or greater than θ, then the vectors are deemed similar.

$$TI_{\alpha,\beta}(r, s) = \frac{|r \cap s|}{|r \cap s| + \alpha|r \setminus s| + \beta|s \setminus r|} \tag{5.1}$$

Shimizu et al. [99] proposed an extension to the Tversky index to securely calculate the similarity between two chemical compound fingerprints represented by

Table 5.1: Notations and acronyms used throughout the chapter.

Symbol	Meaning
θ	Jaccard similarity threshold (0.8 unless stated otherwise)
R (S)	Database relation R (S)
r (s)	Record from relation R (S)
$\overrightarrow{r}$ ($\overrightarrow{s}$)	Bit vector representing record r (s)
ℓ	Length of bit vector
$\|R\|$ ($\|S\|$)	Number of records in relation R (S)
$\|r\|$ ($\|s\|$)	Cardinality of set representing record r (s)
JS	Jaccard similarity
λ	Security parameter
E	Encryption algorithm from the BGV scheme
D	Decryption algorithm from the BGV scheme
m	Plaintext message
q	Modulus for the plaintext space
$\oplus$	Homomorphic addition over ciphertexts
$\otimes$	Homomorphic multiplication over ciphertexts
TI	Tversky index
$\overline{TI}$	Threshold Tversky index
$\alpha, \beta, \gamma, \Gamma, \theta_n, \theta_d$	Threshold Tversky Index parameters
$P_r(x)$ ($P_s(x)$)	Polynomial representing record r (s)
R_G (S_G)	Group of sets from relation R (S)
ψ_{R_G} (ψ_{S_G})	Cardinality of sets in group R_G (S_G)
$\overline{\psi}_{R_G}$ ($\overline{\psi}_{S_G}$)	Upper limit of interval of cardinalities in R_G(S_G)
$\underline{\psi}_{R_G}$ ($\underline{\psi}_{S_G}$)	Lower limit of interval of cardinalities in R_G(S_G)
p	Length of prefix used in the prefix filter

bit vectors $\overrightarrow{r}$ and $\overrightarrow{s}$ (alternatively, $\overrightarrow{r}$ and $\overrightarrow{s}$ can be used to represent sets[1] whose elements are the positions having value 1 in the corresponding bit vector).

They pointed out that disclosing the computed similarity between two chemical compound fingerprints can lead to regression attacks and compromise the protocol's security. To prevent such attacks, the result should only show whether a pair is similar or not, instead of the pair's similarity value. Based on this premise, Shimizu et al. [99] defined the threshold Tversky index ($\overline{TI}_{\alpha,\beta,\theta}$):

Definition 3. *Given the parameters* $\alpha = \mu_a/\gamma, \beta = \mu_b/\gamma$ *and the similarity threshold*

[1]By abuse of the notation used to represent records, we use r and s to denote such sets.

$\theta = \theta_n/\theta_d$, the threshold Tversky Index $\overline{TI}_{\alpha,\beta,\theta}$ between r and s is defined as:

$$\overline{TI}_{\alpha,\beta,\theta}(r,s) = \Gamma|r \cap s| - \theta_n(\mu_a|r| + \mu_b|s|) \tag{5.2}$$

where $\Gamma = (\theta_d - \theta_n)\gamma + \theta_n(\mu_a + \mu_b)$.

Based on this definition, Lemma 1 shows how the threshold Tversky index provides a way to determine whether two data items are similar without disclosing the similarity value itself.

Lemma 1. *If the Tversky index of a pair is greater than or equal to a threshold θ, then the threshold Tversky index using the same parameters is non-negative.*

Proof.

$$TI_{\alpha,\beta}(r,s) \geq \theta$$

$$\frac{|r \cap s|}{|r \cap s| + \alpha|r\ s| + \beta|s\ r|} \geq \theta$$

$$\frac{|r \cap s|}{|r \cap s| + \alpha|r| - \alpha|r \cap s| + \beta|s| - \beta|r\ s|} \geq \theta$$

$$\frac{|r \cap s|}{|r \cap s| + (1 - \alpha - \beta) + \alpha|r|\beta|s|} \geq \theta$$

Considering the parameters, $\alpha = \frac{\mu_a}{\gamma}, \beta = \frac{\mu_b}{\gamma}, \theta = \frac{\theta_n}{\theta_d}$, then:

$$\frac{|r \cap s|}{|r \cap s| + (1 - \frac{\mu_a}{\gamma} - \frac{\mu_a}{\gamma}) + \frac{\mu_a}{\gamma}|r|\frac{\mu_b}{\gamma}|s|} \geq \frac{\theta_n}{\theta_d}$$

$$\theta_d|r \cap s| \geq \theta_n\left(|r \cap s|\left(1 - \frac{\mu_a}{\gamma} - \frac{\mu_b}{\gamma}\right) + \frac{\mu_a}{\gamma}|r| + \frac{\mu_b}{\gamma}|s|\right)$$

$$\theta_d|r \cap s| - \theta_n|r \cap s|\left(1 - \frac{\mu_a}{\gamma} - \frac{\mu_b}{\gamma}\right) - \theta_n\frac{\mu_a}{\gamma}|r| - \theta_n\frac{\mu_b}{\gamma}|s| \geq 0$$

$$\gamma\theta_d|r \cap s| - \theta_n|r \cap s|(\gamma - \mu_a - \mu_b) - \theta_n\mu_a|r| - \theta_n\mu_b|s| \geq 0$$

$$(\gamma\theta_d - \gamma\theta_n + \mu_a\theta_n + \mu_b\theta_n)|r \cap s| - \theta_n(\mu_a|r| + \mu_b|s|) \geq 0$$

$$|r \cap s|((\theta_d - \theta_n)\gamma + \theta_n(\mu_a + \mu_b)) - \theta_n(\mu_a|r| + \mu_b|s|) \geq 0$$

As stated by Def. 3, assuming $\Gamma = (\theta_d - \theta_n)\gamma + \theta_n(\mu_a + \mu_b)$, then:

$$\Gamma|r \cap s| - \theta_n(\mu_a|r| + \mu_b|s|) \geq 0$$

$$\overline{TI}_{\alpha,\beta,\theta}(r,s) \geq 0$$

$\square$

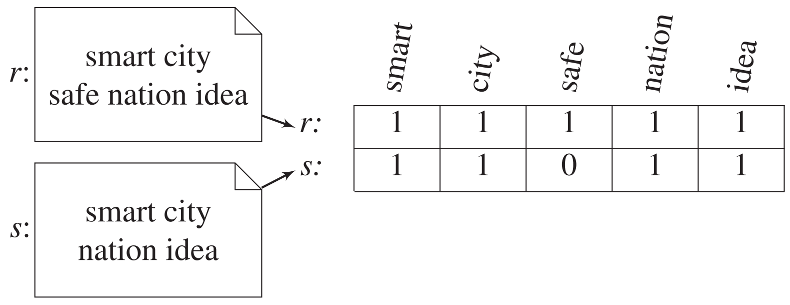


Figure 5.1: Representation of text documents using bit vectors.

Although Shimizu et al. [99] used the threshold Tversky index to calculate the similarity between chemical compound fingerprints, the method can be applied to other kinds of data represented using bit vectors. For example, Fig. 5.1 shows documents (text data) that can be represented by sets of words and then mapped to bit vectors. Each position of the bit vector represents a word found in a collection of documents (i.e., the universe set), and that bit position has the value 1 if that word is in the document corresponding to that vector having such a representation. Then, it is possible to use the threshold Tversky index to calculate the similarity between the documents. Considering a similarity threshold $\theta = 0.8$ ($\theta_n/\theta_d = 8/10 = 4/5$), $\alpha = \beta = \gamma = 1$ ($\mu_a = \mu_b = 1$) and $\Gamma = 9$: $\overline{TI}_{1,1,0.8}(r, s) = 9 \cdot 4 - 4 \cdot (5 + 4) = 0$. Since $\overline{TI}_{1,1,0.8}(r, s) \geq 0$, r and s are considered similar, which can be confirmed by their Jaccard similarity index (JS): $JS(r, s) = |r \cap s|/|r \cup s| = 4/5$.

5.3 Secure Similarity Joins

As mentioned in Sec. 2.5, one way to preserve the privacy of the data processed by similarity joins is to encrypt it before doing the processing. Among the possible encryption methods, probabilistic cryptosystems offer strong security guarantees, being resistant to a number of statistical attacks. On the other hand, they impose limitations regarding the kind of operations that can be done using probabilistic encrypted data.

In particular, as the same message is encrypted to different ciphertexts with high probability, probabilistic cryptosystems make it difficult to perform comparisons. In the case of similarity joins, it is desirable find records from different relations that are more similar than a given threshold. Since the similarity calculation involves comparisons, it cannot be done straightforwardly over data encrypted using probabilistic encryption schemes. Nonetheless, by using probabilistic cryptosystems with homomorphic properties, it is possible to compute the similarity between encrypted records.

5.3.1 Security Model

Our proposal addresses the problem of securely executing similarity joins in an untrusted environment, focusing on preserving the privacy of the relations that are involved in the join. The protocol performs secure similarity joins has two honest-but-curious parties: Alice and Bob. Their objective is to discover what data items they have in common, while keeping the remainder of their datasets private. In addition, in order to prevent regression attacks, it is also important to protect the calculated similarity values themselves [99]. Considering these security requirements, the relations are encrypted using a probabilistic cryptosystem, thereby being protected against frequency attacks.

The protocol is illustrated in Fig. 5.2. After agreeing on the initialization parameters, without loss of generality, suppose that Alice generates the private and public key pair (Sec. 5.3.2). She shares the public key with Bob, so he can encrypt his data with the same key used by Alice to encrypt hers. Alice encrypts her dataset (Sec. 5.3.3) and sends the ciphertexts to Bob, who does the computation of the similarity between Alice's and his data. Bob calculates the similarities between ciphertext pairs using the threshold Tversky index (Sec. 5.3.4) and returns to Alice the ciphertexts corresponding to the threshold Tversky index values. Alice uses her private key to decrypt these values (Sec. 5.3.7) and checks which values are non-negative, which represent pairs that have similarity greater than the threshold specified in the initialization phase. Finally, she shares these results with Bob. At the end of the protocol, both parties have the result of the set similarity join (i.e., pairs of sets whose similarity is greater than the specified threshold) without learning the similarity values themselves.

5.3.2 Initialization

The initialization step consists of choosing the parameters used in the cryptosystem, such as the security parameter λ and the modulus for the plaintext space q. After that, Alice generates the private and public key pair. Also, in this step, the parties select an appropriate similarity threshold for the application.

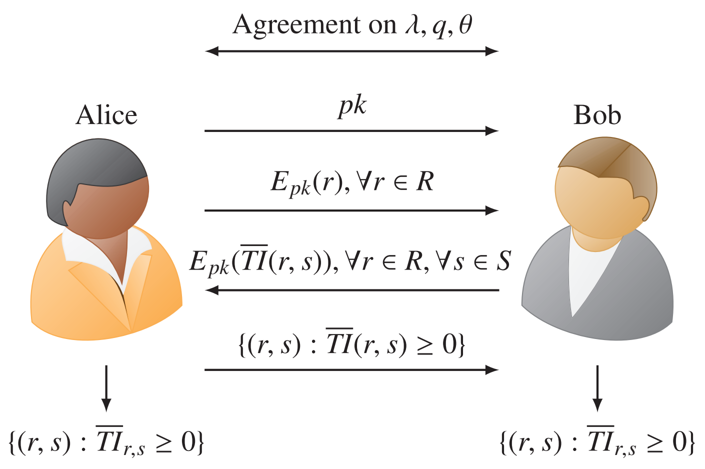

Agreement on λ, q, θ
Alice
pk
Bob
$E_{pk}(r), \forall r \in R$
$E_{pk}(\overline{TI}(r, s)), \forall r \in R, \forall s \in S$
$\{(r, s) : \overline{TI}(r, s) \geq 0\}$
$\{(r, s) : \overline{TI}_{r,s} \geq 0\}$
$\{(r, s) : \overline{TI}_{r,s} \geq 0\}$

Figure 5.2: Secure two-party set similarity join protocol.

5.3.3 Encryption

As explained in Sec. 5.3.4, we assume that each record in the relations is a set represented as a bit vector. Although it would be possible to create one ciphertext per bit, our solution takes advantage of the encryption method provided by the BGV scheme to create a single ciphertext that encrypts a whole bit vector.

More specifically, this method allows all bits of a vector to be embedded into a polynomial, and the polynomial to be encrypted as a single ciphertext. For example, considering r in Fig. 5.1, we obtain the following polynomial $P_r(x) = 1 + x + x^2 + x^3 + x^4$ by using the bits of r as coefficients of $P_r(x)$. After that, $P_r(x)$ is encrypted as $E(P_r(x))$ following the BGV encryption scheme [8].

Although the limit of bits packed can be adjusted according to the parameters chosen in the initialization, in some situations the size of the universe to which the sets belong to can be very large (e.g., $\ell = 2^{64}$ [33]). In such cases, techniques to construct more compact representations of the sets may be used [9]. We highlight that the choice of which technique to use does not affect the security of the protocol, since the bit are encrypted after the bit vector is constructed.

In addition, since there will be less ciphertexts, this method allows us to reduce the memory used, while keeping the same security level as encrypting the bits separately. It also supports a simpler way to calculate the similarity, as explained in Sec. 5.3.4.

Algorithm 3: Calculation of the encrypted threshold Tversky index.

```
input : $E(P_r(x)), E(P_s(x)), E(P_1(x)), E(\Gamma), E(\mu_a), E(\mu_b), E(\theta_n)$
output: $E(\overline{TI}(r,s))$
1 $E(|r \cap s|) \leftarrow E(P_r(x)) \otimes E(P_s(x))$
2 $E(|r|) \leftarrow E(P_r(x)) \otimes E(P_1(x))$
3 $E(|s|) \leftarrow E(P_s(x)) \otimes E(P_1(x))$
4 $E(\overline{TI}_1) \leftarrow E(|r \cap s|) \otimes E(\Gamma)$
5 $E(\overline{TI}_2) \leftarrow E(|r|) \otimes E(\mu_a) \otimes E(\theta_n) \otimes E(-1)$
6 $E(\overline{TI}_3) \leftarrow E(|s|) \otimes E(\mu_b) \otimes E(\theta_n) \otimes E(-1)$
7 $E(\overline{TI}(r,s)) \leftarrow E(\overline{TI}_1) \oplus E(\overline{TI}_2) \oplus E(\overline{TI}_3)$
```

5.3.4 Similarity Calculation

The similarity between two encrypted bit vectors is calculated using the threshold Tversky index (Def. 3). This procedure is shown in Alg. 3, in which the symbols $\oplus$ and $\otimes$ denote homomorphic addition and multiplication operations. Other than operations over the constant parameters $(\Gamma, \theta_n, \mu_a, \mu_b)$, there are three computations done using ciphertexts: $|r \cap s|$, $|r|$, $|s|$.

It is possible to obtain $|r \cap s|$ by calculating the inner product between bit vectors r and s. Although this could be done by a bit-wise multiplication between r and s, followed by a summation of all bits in the resulting bit vector, the encryption method detailed in Sec. 5.3.3 allows for a more efficient way to find the inner product value. We can multiply the polynomial representing the bit vector r by the one representing the bits of s in the inverse order (Alg. 3, line 1). As shown by Lemma 2, the value of the inner product will then be the coefficient of the ℓ-th term of the polynomial resulting from the multiplication, where ℓ is the length of the bit vectors.

Lemma 2. *The inner product between two ℓ-dimensional vectors can be found in the ℓ-th coefficient of a polynomial obtained from the multiplication between the first vector's polynomial representation and the second vector's polynomial representation in the inverse order.*

Proof. Consider two ℓ-dimensional vectors, $a = (a_0, a_1, \dots, a_{\ell-1})$ and $b = (b_0, b_1, \dots, b_{\ell-1})$. We construct a polynomial for each of the vectors using the values in each position as coefficients of the polynomials. In the case of a, we use the values in the order they appear: $P_a(x) = a_0 + a_1 x + \cdots + a_{\ell-1}x^{\ell-1}$. In the case of b, we use the values in the inverse order: $P_b(x) = b_{\ell-1} + b_{\ell-2}x + \cdots + b_0 x^{\ell-1}$.

When multiplying $P_a(x)$ and $P_b(x)$, the terms of the result have the form $a_k x^k \cdot$

$b_k x^{\ell-1-k} = a_k b_k x^{\ell-1}$. Using this fact, we observe that the ℓ-th coefficient (having degree $\ell - 1$) of the resulting polynomial gives the inner product between a and b:

$$\left(\sum_{k=0}^{\ell-1} a_k b_k\right) x^{\ell-1}$$

□

Using the example in Fig. 5.1, consider the polynomial created from r, $P_r(x) = 1+x+x^2+x^3+x^4$, and the inverted polynomial created from s, $P_s(x) = 1+x+x^3+x^4$. In this case, since $\ell = 5$, $|r \cap s|$ is given by the 5th coefficient of $P_r(x) \cdot P_s(x) = 1 + 2x + 2x^2 + 3x^3 + 4x^4 + 3x^5 + 2x^6 + 2x^7 + x^8$, i.e., $|r \cap s| = 4$. In practice, as seen in Alg. 3, this operation is done using the encrypted polynomials.

To find $|r|$, one can just sum the bits of r. However, that would require $\ell - 1$ independent homomorphic additions, which can be reduced to a inner product between r and a bit vector containing only bits 1. Using polynomials, this means multiplying $P_r(x)$ by $P_1(x)$, where $P_1(x)$ is a polynomial having all coefficients equal to 1 (Alg. 3, lines 2–3). The resulting polynomial has the value of the inner product as its ℓ-th coefficient. Still using the same example, $|r|$ would be the 5th coefficient of $P_r(x) \cdot P_1(x) = 1 + 2x + 3x^2 + 4x^3 + 5x^4 + 4x^5 + 3x^6 + 2x^7 + x^8$, i.e., $|r| = 5$. The same method can be used to obtain $|s| = 4$.

After calculating $|r \cap s|$, $|r|$ and $|s|$, the threshold Tversky index value will be in the ℓ-th coefficient of the polynomial resulting from (5.2), as shown in Alg. 3, lines 4–7. In the previous example, considering the same parameters used before (i.e., $\Gamma = 9$, $\theta_n = 4$ and $\mu_a = \mu_b = 1$), $\overline{TI}(r, s)$ is the 5th coefficient of the polynomial $1+2x-2x^2-x^3+0x^4-x^5-2x^6+2x^7+x^8$, i.e., $\overline{TI}(r, s) = 0$. Since the threshold Tversky index between r and s is a non-negative value, this pair is considered similar.

In our protocol, the similarity calculation is performed by Bob using the encrypted data sent by Alice and his own data. Although Bob's data is encrypted with the public key generated by Alice, she does not have access to it, thus being unable to use her secret key to decrypt his data.

5.3.5 Join

The join step determines what pairs should be compared and calculates the similarity for each pair. The most straightforward approach is to compare all combinations of pairs by iterating both relations using a nested loop. We assume the nested-loop join is performed over two relations, R and S, but the same steps are applied for self-joins (i.e., one relation joined with itself).

If R has $|R|$ sets and S has $|S|$ sets, the time complexity of the nested-loop join is $O(|R||S|)$. For this reason, the number of pairs to compare grows quickly as the joined relations become larger. Although in some cases it is necessary to compare all pairs, usually it is possible to apply filtering strategies that can reduce the number of pairs checked [14, 41].

5.3.6 Filtering Strategies

The premise of filtering techniques is that some pairs are too different according to a particular aspect, so their actual comparison can be avoided. The filtering can be based in different aspects, like the difference between cardinalities of sets (i.e., the number of elements in the sets, represented by the number of bits 1 in the bit vectors) or a required minimum overlap between sets. These aspects in particular were used to create the length filter [41] (Sec. 3.1.1) and the prefix filter [14] (Sec. 3.1.1).

We consider the application of these two filter strategies in our work and propose four ways to use them: simple length filter, equi-width length filter, equi-depth length filter and prefix filter. To the best of our knowledge, this is the first work that applies these filters to probabilistically encrypted data by leveraging fully homomorphic encryption.

Simple Length Filter

A simple way to use the length filter is to group sets by cardinality and then calculate what groups would satisfy the conditions given by an adaptation of (3.1). Suppose that R_G and S_G are groups of sets from R and S. For these groups, assume that all the sets in R_G have the same cardinality, ψ_{R_G}. Likewise, all sets in S_G have the same cardinality, ψ_{S_G}. For sets in group R_G to be compared with sets in group S_G, the groups have to satisfy the following conditions:

$$\theta\psi_{S_G} \leq \psi_{R_G} \leq \frac{\psi_{S_G}}{\theta} \tag{5.3}$$

This grouping can be seen in Fig. 5.3. The left table represents relation R, and the right table represents relation S. We omit the values of elements in the sets and show only the cardinalities, which are relevant to the application of length filter. As mentioned, sets having the same cardinality are assigned to the same group.

After grouping, the conditions in (5.3) are checked for each pair of groups in a nested-loop way. Since the number of groups is generally considerably smaller than the number of records, the number of group pairs is usually smaller than the

	Cardinality	Group
r_0	8	R_{G_0}
r_1	7	R_{G_1}
r_2	6	R_{G_2}
r_3	6	
r_4	5	R_{G_3}
r_5	4	R_{G_4}
r_6	1	R_{G_5}
r_7	1	

(a) Relation R

	Cardinality	Group
s_0	9	S_{G_0}
s_1	9	
s_2	8	S_{G_1}
s_3	4	S_{G_2}
s_4	4	
s_5	4	
s_6	2	S_{G_3}
s_7	1	S_{G_4}

(b) Relation S

Figure 5.3: Simple length filter approach, in which sets are grouped according to their cardinality.

number of record pairs. Records in groups that pass the length filter are then joined in the way described in Sec. 5.3.5.

Although disclosing the cardinality values of all records may be tolerable in some scenarios, it might not be acceptable in others that have more strict security requirements. To prevent this information leakage, we execute the length filter in an encrypted way by using homomorphic operations. In this case, the conditions given by (3.1) are adapted to be more easily performed using homomorphic addition and multiplication:

$$\psi_{R_G} - \theta\psi_{S_G} \geq 0 \land \psi_{S_G} - \theta\psi_{R_G} \geq 0 \tag{5.4}$$

By doing so, Bob can calculate the values of the two conditions without knowing the cardinalities of Alice's sets: $E(\psi_{R_G}) \oplus E(-1) \otimes E(\theta) \otimes E(\psi_{S_G})$ and $E(\psi_{S_G}) \oplus E(-1) \otimes E(\theta) \otimes E(\psi_{R_G})$. After that, he sends these values back to Alice so she can decrypt them and check what pairs of groups pass the filter (i.e., pairs whose both conditions yield non-negative values). Alice then tells Bob what groups should be compared, and he calculates the threshold Tversky index for each pair of sets inside the pairs of groups that passed the filter.

For the example, in Fig. 5.3, 6 groups from R will be compared with 5 groups from S, in a total of 30 group pair comparisons. From these, 6 pairs of groups would pass the filter $\{(R_{G_0}, S_{G_0})$, (R_{G_0}, S_{G_1}), (R_{G_1}, S_{G_1}), (R_{G_3}, S_{G_2}), (R_{G_4}, S_{G_2}), $(R_{G_5}, S_{G_4})\}$, yielding a total of 12 pairs of sets whose similarity should be checked. The overall number of comparisons is then $30 + 12 = 42$, meaning a reduction of approximately 35% from the 64 pairs of sets checked by the nested-loop join approach.

As confirmed by the experimental evaluation (Sec. 5.5), the grouping strategy of the simple length filter provides a substantial improvement compared to the nested-loop join. However, when most of the sets have different cardinalities, it is a better idea to group them according to cardinality intervals. By doing so, the groups have upper and lower cardinality bounds, instead of just one value. The intervals can have the same length (equi-width) or the same number of sets in each interval (equi-depth).

Equi-width Length Filter

If many sets have different cardinalities, the number of groups created in the simple length filter approach is close to the number of sets, and the efficacy of the simple length filter is reduced. In such situations, it is possible to divide the whole range of cardinalities into a fixed number of intervals. Figure 5.4 shows sets divided in groups representing intervals of length 2. In this example, R_{G_0} contains sets whose cardinalities are 9 or 8, and R_{G_1} contains sets whose cardinalities are 7 or 6, and so on.

After sets are grouped, we compare pairs of groups and, to pass the filter, a pair has to satisfy the conditions given by (5.5). $\underline{\psi}_{R_G}$ and $\overline{\psi}_{R_G}$ represent the lower and upper bounds of the interval corresponding to group R_G, while $\underline{\psi}_{S_G}$ and $\overline{\psi}_{S_G}$ represent the lower and upper bounds for the group S_G.

$$\overline{\psi}_{R_G} - \theta\underline{\psi}_{S_G} \geq 0 \wedge \overline{\psi}_{S_G} - \theta\underline{\psi}_{R_G} \geq 0 \tag{5.5}$$

In the encrypted version of the equi-width filter, Bob uses the homomorphic properties of the encryption scheme to calculate the values in the left-hand side of the conditions in (5.5): $E(\overline{\psi}_{R_G}) \oplus E(-1) \otimes E(\theta) \otimes E(\underline{\psi}_{S_G})$ and $E(\overline{\psi}_{S_G}) \oplus E(-1) \otimes E(\theta) \otimes E(\underline{\psi}_{R_G})$. As it was done in the encrypted simple length filter, he sends these values to Alice so she can decrypt them and check what pairs have non-negative values for both conditions. After that, Bob proceeds to the calculation of the threshold Tversky index for the sets in the groups that passed the filter.

In the example in Fig. 5.4, 25 pairs of groups are compared, and 8 pairs, $\{(R_{G_0}, S_{G_0}), (R_{G_1}, S_{G_1}), (R_{G_2}, S_{G_2}), (R_{G_2}, S_{G_3}), (R_{G_3}, S_{G_2}), (R_{G_3}, S_{G_3}), (R_{G_4}, S_{G_3}), (R_{G_4}, S_{G_4})\}$, pass the equi-width length filter. These 8 pairs of groups include 20

	Cardinality	Group
r_0	8	R_{G_0}
r_1	7	R_{G_1}
r_2	6	R_{G_1}
r_3	6	R_{G_1}
r_4	5	R_{G_2}
r_5	4	R_{G_2}
r_6	1	R_{G_3}
r_7	1	R_{G_3}

(a) Relation R

	Cardinality	Group
s_0	9	S_{G_0}
s_1	9	S_{G_0}
s_2	8	S_{G_0}
s_3	4	S_{G_1}
s_4	4	S_{G_1}
s_5	4	S_{G_1}
s_6	2	S_{G_2}
s_7	1	S_{G_3}

(b) Relation S

Figure 5.4: Equi-width length filter approach. Sets are divided in intervals of length 2.

pairs of sets that are also compared. In total, 45 comparisons are done using the equi-width length filter (about 30% less than the nested-loop join).

Equi-depth Length Filter

Another way to divide sets is to create groups with the same number of sets. In this case, the cardinality bounds of a group varies depending on the sets in it instead of being fixed, as it happens in the equi-width approach.

For example, Fig. 5.5 shows groups containing two sets each. Since group S_{G_1} has sets whose cardinalities are 4 and 8, its lower bound is $\underline{\psi}_{S_{G_1}} = 4$ and its upper bound is $\overline{\psi}_{S_{G_1}} = 8$.

The comparison between groups uses such lower and upper bounds in the same way as the equi-width approach does, i.e., by checking whether pairs of groups satisfy the conditions in (5.5). The encrypted version of this filter follows the same pattern as the equi-width approach, in which Bob calculates the conditions using encrypted cardinality bounds and sends them for Alice to check the non-negative ones.

In Fig. 5.5, 16 pairs of groups are compared, and 6 pairs, $\{(R_{G_0}, S_{G_0})$, (R_{G_1}, S_{G_1}), (R_{G_1}, S_{G_2}), (R_{G_2}, S_{G_2}), (R_{G_3}, S_{G_2}), $(R_{G_3}, S_{G_3})\}$, pass the filter. From those groups, 24 pairs of sets would be compared, adding to 40 comparisons overall (about 37% less than the nested-loop join).

	Cardinality	Group
r_0	8	R_{G_0}
r_1	7	R_{G_0}
r_2	6	R_{G_1}
r_3	6	R_{G_1}
r_4	5	R_{G_2}
r_5	4	R_{G_2}
r_6	1	R_{G_3}
r_7	1	R_{G_3}

(a) Relation *R*

	Cardinality	Group
s_0	9	S_{G_0}
s_1	9	S_{G_0}
s_2	8	S_{G_1}
s_3	4	S_{G_1}
s_4	4	S_{G_2}
s_5	4	S_{G_2}
s_6	2	S_{G_3}
s_7	1	S_{G_3}

(b) Relation *S*

Figure 5.5: Equi-depth length filter approach. Sets are divided in groups containing 2 sets each.

Prefix Filter

Aspects other than the cardinalities of sets can also be considered when filtering pairs of records. The prefix filter [14] (detailed in Sec. 3.1.1) considers common parts of records and prune pairs which do not share a canonized overlap.

The execution of the prefix filter using sensitive data poses extra challenges. For example, by calculating the prefix in the aforementioned way, not only the elements composing the prefix, but number of such elements can also leak information about the cardinality of the original set.

To address this problem, we propose the encrypted version of the prefix set, exemplified by Fig. 5.6. The initial steps are the same as the plaintext prefix filter, but performed over bit vectors representing sets. Figure 5.6a shows the bit vectors representing the sets, and Fig. 5.6b presents the vectors after being sorted using the alphabetical order.

Then, Alice and Bob construct their prefixes by choosing the first p bits having the value 1, as it can be seen in Fig 5.6c. p is calculated in the same way as the plaintext version.

After that, Bob constructs the inverted index using the prefixes from relation S and compares Alice's prefixes with this index. In practice, the bit vectors are encrypted using the method described in Sec. 5.3.3. Therefore, to check whether the prefixes overlap, Bob calculates the inner product between the bit vectors representing prefixes from R and the bit vectors representing each inverted list in the index. To find the inner products of encrypted vectors, Bob uses the polynomial multiplication, as explained in Sec. 5.3.4.

After computing the inner products, Bob then sends the values so Alice can check which pairs of sets should be actually compared.

Finally, Alice tells Bob what pairs of sets should have their threshold Tversky index calculated, and the protocol enters the join phase (Sec. 5.3.5).

5.3.7 Decryption

As mentioned in Sec. 5.3.3, since Bob does not have the secret key, he must send the values $E(\overline{TI}(r, s))$ for Alice to decrypt. She can then decrypt the ciphertexts and verify whether their value is greater than or equal to zero, which would imply that r and s are similar. Finally, according to the honest-but-curious adversary model, Alice shares that result (i.e., pairs whose threshold Tversky index is non-negative) with Bob.

5.4 Security Analysis

The security of the proposal depends on the security of the BGV encryption scheme [8]. The BGV scheme is based on the Learning with Errors (LWE) problem, which is considered hard even with the advent of quantum computers [73].

The privacy model considers an honest-but-curious adversary that follows the specified protocol, but tries to learn from it. This learning can be done in different ways, including the use of background knowledge that may hint the distribution of sets' cardinalities. However, due to the probabilistic nature of the encryption scheme used, the protocol is resistant to statistical attacks like dictionary and frequency attacks.

Also, as mentioned in Sec. 5.3.3, the method to embed sets into bit vectors do not affect the security, since the bits are independently encrypted after a vector's construction.

In addition, since the use of plaintext filters can leak information regarding the original data, we presented ways to execute the filtering using encrypted values for the cardinalities of the sets and for the prefixes.

	smart	*city*	*safe*	*nation*	*idea*	*secure*	*people*	*country*
r_0:	1	1	1	1	1	0	0	0
r_1:	1	0	1	0	1	1	1	0
s_0:	1	1	0	1	1	0	0	0
s_1:	1	1	1	0	0	0	1	1

(a) Bit vectors representing sets from relations R and S.

	city	*country*	*idea*	*nation*	*people*	*safe*	*secure*	*smart*
r_0:	1	0	1	1	0	1	0	1
r_1:	0	0	1	0	1	1	1	1
s_0:	1	0	1	1	0	0	0	1
s_1:	1	1	0	0	1	1	0	1

(b) Bit vectors representing sets' elements sorted according to alphabetical order.

	city	*country*	*idea*	*nation*	*people*	*safe*	*secure*	*smart*
r_0:	1	0	1	0	0	0	0	0
r_1:	0	0	1	0	1	0	0	0
s_0:	1	0	0	0	0	0	0	0
s_1:	1	1	0	0	0	0	0	0

(c) Bit vectors representing prefixes of each set.

city	*country*	*idea*	*nation*	*people*	*safe*	*secure*	*smart*	
1	0	0	0	0	0	0	0	$\rightarrow s_0, s_1$
0	1	0	0	0	0	0	0	$\rightarrow s_1$

(d) Inverted index built using prefixes from relation S.

Figure 5.6: Example of the encrypted version of prefix filter.

5.5 Experimental Evaluation

This section presents the results of experiments performed to evaluate the proposal. We compared different implementations of the protocol, varying in terms of execution model (i.e., serial or parallel) and filtering approach. Unless stated otherwise, the similarity threshold used was 0.8.

5.5.1 Setup

To validate the applicability of the proposal, the experimental evaluation was done using three real datasets, whose characteristics are detailed in Sec. 4.4.1: paper abstracts (ABSTRACTS), image features (IMAGES) and transactions (TRANSACTIONS).

As described in Sec. 4.4.1, we create relations R and S by randomly choosing sets from the original data sets, varying relations' sizes from 1,024 to 524,288.

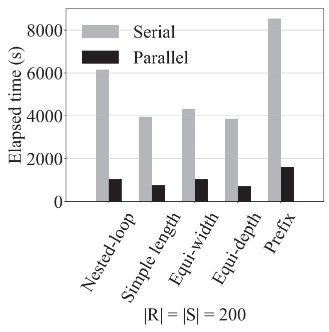
Regarding the construction of the bit vectors, we adopted the same encoding scheme used by Schnell et al. [96]. The bit vectors (Bloom filters) of size $\ell = 1000$ were generated by hashing each element of the set representing each record using ten hash functions in the form $h_i(x) = SHA256(x) + i \cdot MD5(x) \bmod \ell, 1 \leq i \leq 10$. As highlighted in Sec. 5.3.3, this choice of encoding does not affect the security of the protocol, as the bits of the Bloom filters are independently encrypted using the BGV scheme's encryption algorithm.

The server used in the evaluation was an Intel Xeon E5–1650 v2 (6 cores, 12 threads) with 32GB RAM. In terms of software, we used CentOS 6.8, GCC 6.1.0 and the libraries GMP 6.1.1, Crypto++ 5.6.3, NTL 9.10.0 and HElib 1.3.

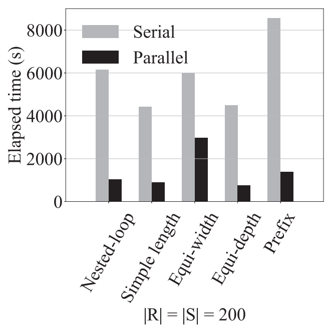
5.5.2 Parallelization Impact on Performance

As previously mentioned, we explore parallelization opportunities in most steps of the protocol, in order to mitigate the computational overhead created by the fully homomorphic cryptosystem used. We parallelized the encryption and decryption phases, as well as the join phase in which records are compared, such that the similarities of different pairs are calculated in parallel.

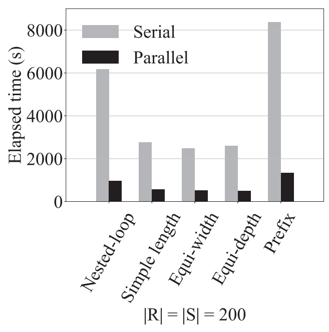
Figure 5.7 shows the impact of using parallelization when executing the nested-loop join and the different filters. The parallel version of most implementations yielded speedups of 4 to 6 times when compared to the serial version, except in the case of the equi-width filter when processing the IMAGES dataset.



(a) ABSTRACTS



(b) IMAGES



(c) TRANSACTIONS

Figure 5.7: Parallelization impact on performance.

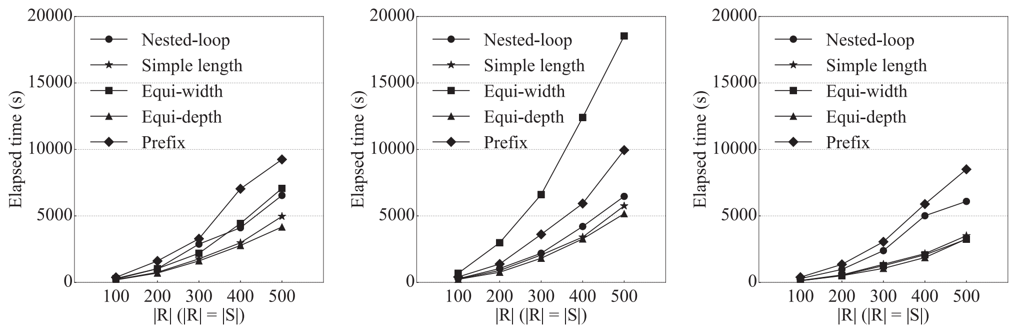
The reason for speedup of only two times in this case was that all the sets in the relation constructed from the IMAGES dataset had the same cardinality. When representing such sets using bit vectors, the number of bits 1 in the vectors is similar, translating into a small number of groups containing a large number of records. Since groups were processed concurrently, a smaller number of groups limited the gains of the parallelized version.

Due to its superior performance, all the following results are related to experiments using the parallel version of all approaches. We observed similar trends using the serial version.

5.5.3 Encrypted Filtering Performance

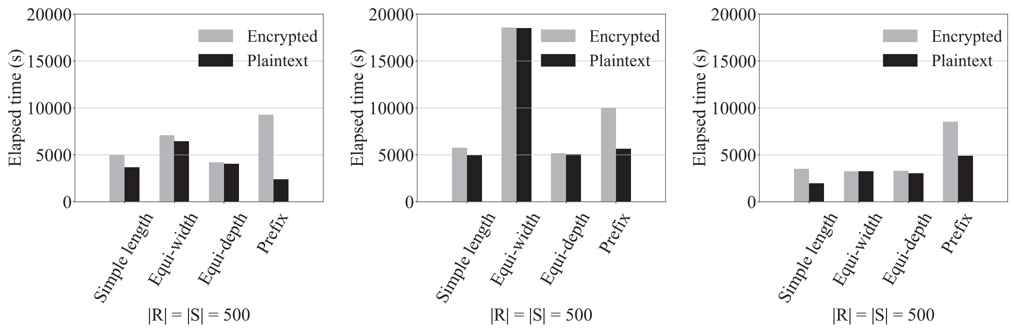
The results depicted in Fig. 5.8 reveal how the performance of the filters changed with varying relations' sizes. Independently of the dataset used, the equi-depth approach was the fastest one thanks to its low cost and high pruning power characteristics.

Interestingly, except in the case of the IMAGES dataset (due to reasons explained in Sec. 5.5.2), the prefix filter had the worst performance. Since extra bit vectors were created to maintain the index, the encrypted prefix filter introduced a high overhead to the filter step, resulting in a considerable increase in the overall execution time.



(a) ABSTRACTS (b) IMAGES (c) TRANSACTIONS

Figure 5.8: Performance of encrypted filters for varying relations' sizes.



(a) ABSTRACTS (b) IMAGES (c) TRANSACTIONS

Figure 5.9: Performance comparison between encrypted and plaintext versions of filters.

5.5.4 Plaintext Filtering Performance

In the cases in which a better performance is desirable in favor of the privacy-preservation of terms used in the filters, using plaintext filters can reduce the cost of the filtering step. As it can be seen from Fig. 5.9, using the plaintext version is mainly effective in the case of prefix filter. As mentioned in Sec. 5.5.3, the encrypted prefix filter has a high overhead, which does not exist in the case of the plaintext version. On the other hand, since not so much time was taken by the encrypted equi-width and equi-depth filters, using their plaintext versions did not offer expressive gains.

5.6 Summary

In this chapter, we have proposed a protocol for secure similarity joins that can be used in untrusted environments to process sensitive data. The protocol uses probabilistic encryption, and it is secure against statistical attacks. All the computation is done over ciphertexts created using fully homomorphic encryption, and the similarity computation uses the threshold Tversky index which also protects the similarity value itself.

Despite the theoretical soundness, current fully homomorphic encryption schemes still have drawbacks in terms of performance, which might hinder their adoption in a large scale. To address this obstacle, we have presented the use of parallel processing and the adaptation of filtering strategies to reduce the number of pairs compared in the join step.

In its current state, the proposed scheme can be used in applications that have high security requirements and that are tolerant of performance penalties. Examples of such applications include the exchange of highly sensitive proprietary data between enterprises, join of genomic data of high profile individuals for research purposes and integration of databases related to law enforcement for the identification of criminals.

The use of GPU have been considered in this solution to improve performance, but the large ciphertext expansion, characteristic of fully homomorphic schemes, makes it impractical to fit more than a few dozens of ciphertexts in the GPUs memory. On the other hand, as discussed in Sec. 6.2, the use of other accelerators that have less memory limitations (e.g., Intel Xeon Phi) may be a viable alternative for faster processing. In addition, in terms of scalability, the parallelism provided by the solution can also be explored by clusters of CPUs in a distributed fashion.

Although in this work we considered the BGV scheme [8], the proposal can be adapted to other schemes, as long as the additive and multiplicative homomorphic properties are available. For this reason, the performance of our protocol can be improved not only by the use of parallelization or filter strategies, but also as faster and more efficient fully homomorphic encryption schemes are developed.

Chapter 6

Conclusion

This chapter concludes this dissertation by summarizing the main contributions of our work and indicating possible avenues for future work.

6.1 Summary of Contributions

The numerous applications of set similarity joins require agility in their execution and, in case of sensitive data, privacy-preserving processing methods. In this dissertation, we have proposed solutions to address such efficiency and security challenges. In particular, we have exploited recent advances in terms of parallel processing by using GPUs as well as the strong security and flexibility provided by fully homomorphic cryptosystems. Based on this, we highlight our main contributions:

- A new scheme to accelerate set similarity joins by exploring parallel processing. We made use of MinHash's storage efficiency to overcome memory challenges posed by GPUs and achieved high speedups in comparison to CPU-only implementations.
- To preserve the privacy of the data being joined, we introduced a two-party protocol for similarity join execution. The protocol is based on fully homomorphic encryption schemes and harness their security properties to protect the data. In order to securely compute whether two sets are similar, we employ threshold Tversky index, which also secures the similarity value itself and safeguards against regression attacks. The performance issues associated with fully homomorphic encryption schemes are addressed by adapting filter strategies to the privacy-preservation context. Such methods decrease the

number of pairs that are compared and help improve the overall execution time.

6.2 Future Work

We have identified opportunities for future work in the following areas:

- Join algorithms and scalability: Since the join is still an expensive part of the processing, future work can explore the adaptation of partitioning techniques and set relations to reduce the number of comparisons and improve performance. The implementation of such techniques using GPUs also require special considerations; i.e., the chosen algorithms should have parallelizable processing-intensive parts and infrequent memory transfers. Another option is to use a higher number of hardware components, like multiple GPUs or clusters containing multiple nodes with CPUs and GPUs.

- Efficiency of the privacy-preserving protocol: As shown in this work, the slow execution of encrypted data processing may hinder its adoption in some scenarios. To deal with this drawback, we suggest the study of dimension reduction techniques with the objective of decreasing the sizes of ciphertexts. Although this creates a trade-off between performance and accuracy, it can help abate the costs imposed by fully homomorphic encryption. The combination of different approaches to achieve higher scalability seems advantageous. For instance, the utilization of a different accelerator, Intel Xeon Phi, can aid in the faster processing of larger relations. This is especially promising when dealing with the large ciphertexts yielded by fully homomorphic cryptosystems.

Bibliography

[1] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *VLDB*, pages 918–929. ACM, 2006.

[2] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *WWW*, pages 131–140. ACM, 2007.

[3] Mihir Bellare, Alexandra Boldyreva, and Adam ONeill. Deterministic and efficiently searchable encryption. *Advances in Cryptology-CRYPTO 2007*, pages 535–552, 2007.

[4] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[5] Christian Böhm, Robert Noll, Claudia Plant, and Andrew Zherdin. Index-supported similarity join on graphics processors. In *BTW*, volume 144, pages 57–66, 2009.

[6] Dan Boneh and Brent Waters. Conjunctive, subset, and range queries on encrypted data. *Theory of cryptography*, pages 535–554, 2007.

[7] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. Spatio-textual similarity joins. *Proceedings of the VLDB Endowment*, 6(1):1–12, 2012.

[8] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3):13:1–13:36, July 2014.

[9] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *J. Comput. Syst. Sci.*, 60(3):630–659, 2000.

[10] Andrei Z Broder, Steven C Glassman, Mark S Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8-13):1157–1166, 1997.

[11] Ning Cao, Cong Wang, Ming Li, Kui Ren, and Wenjing Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Transactions on parallel and distributed systems*, 25(1):222–233, 2014.

[12] Kaushik Chakrabarti, Surajit Chaudhuri, Venkatesh Ganti, and Dong Xin. An efficient filter for approximate membership checking. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 805–818. ACM, 2008.

[13] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ACNS*, volume 5, pages 442–455. Springer, 2005.

[14] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *ICDE*, page 5. IEEE Computer Society, 2006.

[15] Peter Christen, Rainer Schnell, Dinusha Vatsalan, and Thilina Ranbaduge. Efficient cryptanalysis of bloom filters for privacy-preserving record linkage. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 628–640. Springer, 2017.

[16] Chris Clifton, Murat Kantarcioglu, Jaideep Vaidya, Xiaodong Lin, and Michael Y Zhu. Tools for privacy preserving distributed data mining. *ACM Sigkdd Explorations Newsletter*, 4(2):28–34, 2002.

[17] William W Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *ACM SIGMOD Record*, volume 27, pages 201–212. ACM, 1998.

[18] William W Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Transactions on Information Systems (TOIS)*, 18(3):288–321, 2000.

[19] Ana Costache and Nigel P Smart. Which ring based somewhat homomorphic encryption scheme is best? In *Cryptographers' Track at the RSA Conference*, pages 325–340. Springer, 2016.

[20] Helei Cui, Xingliang Yuan, Yifeng Zheng, and Cong Wang. Enabling secure and effective near-duplicate detection over encrypted in-network storage. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pages 1–9. IEEE, 2016.

[21] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5):895–934, 2011.

[22] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[23] Abhinandan S Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th international conference on World Wide Web*, pages 271–280. ACM, 2007.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[25] Dong Deng, Guoliang Li, Shuang Hao, Jiannan Wang, and Jianhua Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, 2014.

[26] Dong Deng, Guoliang Li, He Wen, and Jianhua Feng. An efficient partition based method for exact set similarity joins. *Proceedings of the VLDB Endowment*, 9(4):360–371, 2015.

[27] Lee R Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[28] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of data integration*. Elsevier, 2012.

[29] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike J. Sloan, Charles Boyd, and John Manferdelli. Fast scan algorithms on graphics processors. In Pin Zhou, editor, *ICS*, pages 205–213. ACM, 2008.

[30] L Dusserre, C Quantin, and H Bouzelat. A one way public key cryptosystem for the linkage of nominal files in epidemiological studies. *Medinfo. MEDINFO*, 8:644–647, 1995.

[31] Cynthia Dwork. Differential privacy: A survey of results. In *International Conference on Theory and Applications of Models of Computation*, pages 1–19. Springer, 2008.

[32] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[33] Dennis Fetterly, Mark Manasse, Marc Najork, and Janet L Wiener. A large-scale study of the evolution of web pages. *Software: Practice and Experience*, 34(2):213–237, 2004.

[34] NIST FIPS. 180-2: Secure hash standard (shs). *Information Technology Laboratory, National Institute of Standards and Technology (October 2008), http://csrc. nist. gov/publications/fips/fips180-3/fips180-3_final. pdf*, 2001.

[35] Zhangjie Fu, Xinle Wu, Chaowen Guan, Xingming Sun, and Kui Ren. Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement. *IEEE Transactions on Information Forensics and Security*, 11(12):2706–2716, 2016.

[36] Craig Gentry et al. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.

[37] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. Cryptology ePrint Archive, Report 2013/340, 2013. `http://eprint.iacr.org/2013/340`.

[38] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 365–377. ACM, 1982.

[39] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[40] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 89–98. Acm, 2006.

[41] Luis Gravano, Panagiotis G. Ipeirotis, Hosagrahar Visvesvaraya Jagadish, Nick Koudas, Shanmugauelayut Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of the 27th International Conference on Very Large Databases (VLDB 2001)*, pages 491–500, 2001.

[42] Joseph L Greathouse and Mayank Daga. Efficient sparse matrix-vector multiplication on gpus using the csr storage format. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 769–780. IEEE Press, 2014.

[43] Rob Hall and Stephen E Fienberg. Privacy-preserving record linkage. In *Privacy in statistical databases*, volume 6344, pages 269–283. Springer, 2010.

[44] Isabelle Hang, Florian Kerschbaum, and Ernesto Damiani. Enki: Access control for encrypted query processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 183–196, New York, NY, USA, 2015. ACM.

[45] Mark Harris. Parallel prefix sum (scan) with cuda, 2009.

[46] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, December 2009.

[47] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[48] Monika Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 284–291. ACM, 2006.

[49] Timothy C Hoad and Justin Zobel. Methods for identifying versioned and plagiarized documents. *Journal of the Association for Information Science and Technology*, 54(3):203–215, 2003.

[50] Jared Hoberock and Nathan Bell. *Thrust: A Productivity-Oriented Library for CUDA*. 2012.

[51] Anna Huang. Similarity measures for text document clustering. In *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, pages 49–56, 2008.

[52] Ali Inan, Murat Kantarcioglu, Elisa Bertino, and Monica Scannapieco. A hybrid approach to private record linkage. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 496–505. IEEE, 2008.

[53] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[54] Paul Jaccard. The distribution of the flora in the alpine zone. *New phytologist*, 11(2):37–50, 1912.

[55] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. String similarity joins: An experimental evaluation. *Proc. VLDB Endow.*, 7(8):625–636, April 2014.

[56] Mark Johnson, Prakash Ishwar, Vinod Prabhakaran, Daniel Schonberg, and Kannan Ramchandran. On compressing encrypted data. *IEEE Transactions on Signal Processing*, 52(10):2992–3006, 2004.

[57] Alexandros Karakasidis, Vassilios Verykios, and Peter Christen. Fake injection strategies for private phonetic matching. *Data Privacy Management and Autonomous Spontaneus Security*, pages 9–24, 2012.

[58] Florian Kerschbaum. Privacy-preserving computation. In *Annual Privacy Forum*, pages 41–54. Springer, 2012.

[59] Martin Kroll and Simone Steinmetzer. Who is 1011011111... 1110110010? automated cryptanalysis of bloom filter encryptions of databases with several personal identifiers. In *International Joint Conference on Biomedical Engineering Systems and Technologies*, pages 341–356. Springer, 2015.

[60] Mehmet Kuzu, Mohammad Saiful Islam, and Murat Kantarcioglu. Efficient similarity search over encrypted data. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1156–1167. IEEE, 2012.

[61] Mehmet Kuzu, Murat Kantarcioglu, Elizabeth Ashley Durham, Csaba Toth, and Bradley Malin. A practical approach to achieve private medical record linkage in light of public resources. *Journal of the American Medical Informatics Association*, 20(2):285–292, 2013.

[62] Scott T Leutenegger and Daniel Dias. *A modeling study of the TPC-C benchmark*, volume 22. ACM, 1993.

[63] Fengjun Li, Yuxin Chen, Bo Luo, Dongwon Lee, and Peng Liu. Privacy preserving group linkage. In *International Conference on Scientific and Statistical Database Management*, pages 432–450. Springer, 2011.

[64] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[65] Jin Li, Qian Wang, Cong Wang, Ning Cao, Kui Ren, and Wenjing Lou. Fuzzy keyword search over encrypted data in cloud computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–5. IEEE, 2010.

[66] Ping Li and Arnd Christian Knig. b-bit minwise hashing. *CoRR*, abs/0910.3349, 2009.

[67] Ping Li, Art B. Owen, and Cun-Hui Zhang. One permutation hashing for efficient search and learning. *CoRR*, abs/1208.1259, 2012.

[68] Ping Li, Anshumali Shrivastava, and Arnd Christian König. Gpu-based minwise hashing: Gpu-based minwise hashing. In *Proceedings of the 21st World Wide Web Conference (WWW 2012) (Companion Volume)*, pages 565–566, 2012.

[69] Michael D. Lieberman, Jagan Sankaranarayanan, and Hanan Samet. A fast similarity join algorithm using graphics processing units. In Gustavo Alonso, Jos A. Blakeley, and Arbee L. P. Chen, editors, *ICDE*, pages 1111–1120. IEEE, 2008.

[70] Yehuda Lindell and Benny Pinkas. Secure multiparty computation for privacy-preserving data mining. *Journal of Privacy and Confidentiality*, 1(1):5, 2009.

[71] Adriana Lopez-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. Cryptology ePrint Archive, Report 2013/094, 2013. `http://eprint.iacr.org/2013/094`.

[72] Ji-zhou Luo, Sheng-fei Shi, Hong-zhi Wang, and Jian-zhong Li. Frepjoin: an efficient partition-based algorithm for edit similarity join. *Frontiers of Information Technology & Electronic Engineering*, 18(10):1499–1510, Oct 2017.

[73] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.

[74] Willi Mann and Nikolaus Augsten. Pel: Position-enhanced length filter for set similarity joins. In *Grundlagen von Datenbanken*, pages 89–94, 2014.

[75] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. An empirical evaluation of set similarity join techniques. *Proceedings of the VLDB Endowment*, 9(9):636–647, 2016.

[76] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Detectives: detecting coalition hit inflation attacks in advertising networks streams. In *Proceedings of the 16th international conference on World Wide Web*, pages 241–250. ACM, 2007.

[77] Ahmed Metwally and Christos Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *PVLDB*, 5(8):704–715, 2012.

[78] Noman Mohammed, Benjamin C Fung, and Mourad Debbabi. Anonymity meets game theory: secure data integration with malicious participants. *The VLDB JournalThe International Journal on Very Large Data Bases*, 20(4):567–588, 2011.

[79] AB MySQL. Mysql: the world's most popular open source database. *http://www. mysql. com/*, 2005.

[80] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. 2007.

[81] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, 2013.

[82] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[83] Pascal Paillier. *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes*, pages 223–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[84] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.

[85] Manoj M Prabhakaran and Amit Sahai. *Secure multi-party computation*, volume 10. IOS press, 2013.

[86] Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1033–1044. ACM, 2011.

[87] Vernica Rares, Carey Michael J., and Li Chen. Efcient parallel set-similarity joins using mapreduce. 2010.

[88] Leonardo Andrade Ribeiro and Theo Hrder. Generalizing prefix filtering to improve set similarity joins. *Inf. Syst.*, 36(1):62–78, 2011.

[89] Sidney Ribeiro-Junior, Rafael David Quirino, Leonardo Andrade Ribeiro, and Wellington Santos Martins. Fast parallel set similarity joins on many-core architectures. *Journal of Information and Data Management*, 8(3):255, 2017.

[90] Ronald Rivest. The md5 message-digest algorithm. 1992.

[91] Mehran Sahami and Timothy D Heilman. A web-based kernel function for measuring the similarity of short text snippets. In *Proceedings of the 15th international conference on World Wide Web*, pages 377–386. AcM, 2006.

[92] Sunita Sarawagi and Alok Kirpal. Efficient set joins on similarity predicates. In Gerhard Weikum, Arnd Christian Knig, and Stefan Deloch, editors, *SIGMOD Conference*, pages 743–754. ACM, 2004.

[93] Muhammad I. Sarfraz, Mohamed Nabeel, Jianneng Cao, and Elisa Bertino. Dbmask: Fine-grained access control on encrypted relational databases. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 1–11, New York, NY, USA, 2015. ACM.

[94] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295. ACM, 2001.

[95] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *International Workshop on Fast Software Encryption*, pages 191–204. Springer, 1993.

[96] Rainer Schnell, Tobias Bachteler, and Jörg Reiher. Privacy-preserving record linkage using bloom filters. *BMC medical informatics and decision making*, 9(1):41, 2009.

[97] Ziad Sehili, Lars Kolb, Christian Borgs, Rainer Schnell, and Erhard Rahm. Privacy preserving record linkage with ppjoin. In *BTW*, pages 85–104, 2015.

[98] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In Mark Segal and Timo Aila, editors, *Graphics Hardware*, pages 97–106. Eurographics Association, 2007.

[99] Kana Shimizu, Koji Nuida, Hiromi Arai, Shigeo Mitsunari, Nuttapong Attrapadung, Michiaki Hamada, Koji Tsuda, Takatsugu Hirokawa, Jun Sakuma, Goichiro Hanaoka, et al. Privacy-preserving search for chemical compound databases. *BMC bioinformatics*, 16(Suppl 18):S6, 2015.

[100] Nigel P Smart. *Cryptography made simple*. Springer, 2016.

[101] Nigel P Smart. Secure multi-party computation. In *Cryptography Made Simple*, pages 439–450. Springer, 2016.

[102] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 44–55. IEEE, 2000.

[103] Ellen Spertus, Mehran Sahami, and Orkut Buyukkokten. Evaluating similarity measures: a large-scale study in the orkut social network. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 678–684. ACM, 2005.

[104] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. *Proc. VLDB Endow.*, 6(5):289–300, March 2013.

[105] Dinusha Vatsalan, Peter Christen, and Erhard Rahm. Scalable multi-database privacy-preserving record linkage using counting bloom filters. *arXiv preprint arXiv:1701.01232*, 2017.

[106] Dinusha Vatsalan, Peter Christen, and Vassilios S Verykios. An efficient two-party protocol for approximate matching in private record linkage. In *Proceedings of the Ninth Australasian Data Mining Conference-Volume 121*, pages 125–136. Australian Computer Society, Inc., 2011.

[107] Dinusha Vatsalan, Peter Christen, and Vassilios S Verykios. A taxonomy of privacy-preserving record linkage techniques. *Information Systems*, 38(6):946–969, 2013.

[108] Bing Wang, Shucheng Yu, Wenjing Lou, and Y Thomas Hou. Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud. In *INFOCOM, 2014 Proceedings IEEE*, pages 2112–2120. IEEE, 2014.

[109] Cong Wang, Kui Ren, Shucheng Yu, and Karthik Mahendra Raje Urs. Achieving usable and privacy-assured similarity search over outsourced cloud data. In *INFOCOM, 2012 Proceedings IEEE*, pages 451–459. IEEE, 2012.

[110] Jiannan Wang, Tim Kraska, Michael J Franklin, and Jianhua Feng. Crowder: Crowdsourcing entity resolution. *Proceedings of the VLDB Endowment*, 5(11):1483–1494, 2012.

[111] Jiannan Wang, Guoliang Li, and Jianhua Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In K. Seluk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *SIGMOD Conference*, pages 85–96. ACM, 2012.

[112] Jiannan Wang, Guoliang Li, Tim Kraska, Michael J Franklin, and Jianhua Feng. Leveraging transitive relations for crowdsourced joins. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 229–240. ACM, 2013.

[113] Wei Wang, Jianbin Qin, Chuan Xiao, Xuemin Lin, and Heng Tao Shen. Vchunkjoin: An efficient algorithm for edit similarity joins. *IEEE Trans. Knowl. Data Eng.*, 25(8):1916–1929, 2013.

[114] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. Leveraging set relations in exact set similarity join. *Proceedings of the VLDB Endowment*, 10(9):925–936, 2017.

[115] William E Winkler. The state of record linkage and current research problems. In *Statistical Research Division, US Census Bureau*. Citeseer, 1999.

[116] Zhihua Xia, Xinhui Wang, Xingming Sun, and Qian Wang. A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):340–352, 2016.

[117] Chuan Xiao, Wei Wang 0011, and Xuemin Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[118] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 131–140, New York, NY, USA, 2008. ACM.

[119] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)*, 36(3):15, 2011.

[120] Cairong Yan, Jian Wang, Bin Zhu, and Wenjing Guo. Para-join: an efficient parallel method for string similarity join. *International Journal of High Performance Computing and Networking*, 10(4-5):381–390, 2017.

[121] Shengen Yan, Guoping Long, and Yunquan Zhang. Streamscan: fast scan algorithms for gpus without global barrier synchronization. In Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard Vuduc, editors, *PPOPP*, pages 229–238. ACM, 2013.

[122] Byoungju Yang, Jaeseok Myung, Sang-goo Lee, and Dongjoo Lee. A mapreduce-based filtering algorithm for vector similarity join. In *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '13, pages 71:1–71:5, New York, NY, USA, 2013. ACM.

[123] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.

[124] Xingliang Yuan, Helei Cui, Xinyu Wang, and Cong Wang. Enabling privacy-assured similarity retrieval over millions of encrypted records. In *European Symposium on Research in Computer Security*, pages 40–60. Springer, 2015.

[125] Xingliang Yuan, Xinyu Wang, Cong Wang, Chenyun Yu, and Sarana Nutanong. Privacy-preserving similarity joins over encrypted data. *IEEE Transactions on Information Forensics and Security*, 12(11):2763–2775, 2017.

List of Publications

Refereed journal papers

- Mateus S. H. Cruz, Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa, “Probabilistic Frequent Itemset Mining on a GPU Cluster” *Transactions on Large-Scale Data and Knowledge-Centered Systems XXVIII: Special Issue on Database and Expert Systems Applications*, pp. 1–22, September 2016.

Refereed international conference papers

- Mateus S. H. Cruz, Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa, “GPU Acceleration of Set Similarity Joins” in *Proceedings of the 26th International Conference on Database and Expert Systems Applications (DEXA)*, pp. 384–398, Valencia, Spain, September 1–4, 2015.
- Mateus S. H. Cruz, Toshiyuki Amagasa, Chiemi Watanabe, Wenjie Lu, and Hiroyuki Kitagawa, “Secure Similarity Joins Using Fully Homomorphic Encryption” in *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services (iiWAS)*, pp. 224–233, Salzburg, Austria, December 4–6, 2017.