Leveraging Positive and Negative Representations of Information

by

Eric Delarosa Trias

B.S., Computer Science, University of California, Davis, 1988

M.S., Computer Engineering, Air Force Institute of Technology, 2002

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2008

Dedication

Sa aking pamilia, mga kaibigan at guro na tumulong para ako'y magtagumpay.

Acknowledgments

First and foremost, I thank God for his countless blessings and for giving me the strength to persevere. I also know that I couldn't have done this without the support of my wife and children. They sacrificed much for me to finish.

I was fortunate to have dedicated collaborators who were essential to this research. I would like to thank Fernando Esponda for his help and inspiration. To Elena S. Ackley, your enthusiasm and hard work gave life to our research. To Jorge Navas, your partnership was invaluable to this work.

I heartily acknowledge Prof. Stephanie Forrest, my advisor and dissertation chair, for her guidance and encouragement. I also thank my committee members, Prof. Greg Heileman, Prof. Terran Lane, and Prof. Jedidiah Crandall.

Finally, I thank the U.S. Air Force and the Air Force Institute of Technology for providing this fellowship opportunity. The views expressed in this article are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

Leveraging Positive and Negative Representations of Information

by

Eric Delarosa Trias

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2008

Leveraging Positive and Negative Representations of Information

by

Eric Delarosa Trias

B.S., Computer Science, University of California, Davis, 1988

M.S., Computer Engineering, Air Force Institute of Technology, 2002

Ph.D., Computer Science, University of New Mexico, 2008

Abstract

In this dissertation, I advance the concept known as *negative databases*, or negative representation of information, by developing a negative relational algebra, a negative database management system architecture, and by exploring how negative information can improve finding solutions to combinatorial problems. This dissertation shows how to combine positive data with its complement, known as negative data, to provide secure and/or compact representation as required by an application.

Positive data consist of a set of binary strings of a given length l. A negative database represents this set by maintaining a set of ternary strings matching all strings of length l, except those in the positive set. It is known that certain negative databases are $\mathcal{NP}$-Hard to reverse, in which case an adversary must resort to enumerating all 2^l strings to recover the positive set. Although recovering of positive

data is difficult, answering membership queries remains efficient — an advantage for privacy-enhancing applications. This dissertation extends the negative database concept by defining an algebra that corresponds to the positive relational model. Understanding these operations and their complexity illuminates the complexity of queries and set operations over negative databases. Certain operations are shown to be more efficient than their positive counterparts and others are shown to be less efficient. In addition, I develop a negative database architecture that uses a traditional relational database management system to better manage both positive and negative data simultaneously. By bringing negative databases into a more mainstream concept, others will more easily understand and adopt them for their use. Lastly, I apply negative database ideas to abstract interpretation, more specifically, set-sharing analysis of logic programs. For a given number of program variables, the number of variable sharings can be exponential leading to an intractable problem instance. I show how to represent set-sharings more efficiently and precisely by maintaining a compact positive or negative representation, as required by the analysis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's technology and communication infrastructure provides unlimited access to vast amounts of data. Such ubiquitous access to information has privacy advocates clamoring for more control over who has access to information and how that information is used. This concern is countered by the increasing need to collaborate and share information, fueling a tendency to make available as much data as possible. However, responsible data owners, motivated by prudent business practices or government regulations, strive to control access to sensitive data. They are more likely to share data if privacy protection were guaranteed and/or if they had control over the types of operations conducted over their data. However, these privacy enhancing protocols introduce many system challenges especially in handling sensitive data from disparate sources.

Privacy enhancing applications are in demand and have been studied by several researchers [3, 8, 18, 50, 52, 69, 53, 106]. Cryptography has been the most common primitive for developing solutions to privacy enhancing applications. Current cryptographic protocols are based on number theoretic problems such as factoring, discrete logarithm, and elliptic logarithm [37]. These problems are related so if one is broken

then all may be compromised [36]. I explore using an alternative tool called negative databases, which is based on the difficulty of finding satisfying (SAT) assignments to Boolean logic formulas. Another disadvantage of using encryption is that key management, to include key distribution, overhead may not be appropriate for certain applications. To our favor, negative databases do not depend on the concept of keys for its security. One of the primary goals of this research is to establish negative databases as a viable alternative to encryption for privacy-enhancing applications.

Negative representation of information, as proposed by Fernando Esponda, attempts to address data privacy requirements without cryptographic means. It was inspired by the theory of self-nonself discrimination of the immune system [38]. Immune cells recognize the body's own cells and molecules from foreign ones. These detectors, called T-cells, undergo a filtering process in the thymus, called negative selection, which destroys T-cells that recognize itself. Thus, only T-cells that detects foreign antigens remain and are distributed throughout the body. This process enables the immune system to detect antigens without initially encountering them. From an information content perspective, the collection of active T-cells, those trained to recognize nonself, can be viewed, in complement form, as a model of what self truly is. He proposed a mapping of nonself representation as a set of strings enabling them to be treated as a database. Taken from a finite universe, this negative database is a representation of all the elements *not* contained in a given positive set. Although possessing different properties, a set and its complement contain the same amount of information. An important observation is that queries can be answered by either a positive or negative database by complementing the results accordingly. For example, if we ask whether a string is in the positive database using its negative representation and we get a `false` answer (does not appear in the negative database), then we know the string is actually in the positive set and the answer is `true`. However, as a privacy-enhancing feather of the negative representation, the positive information is not apparent and can be difficult to retrieve. In this way,

negative representations can potentially be used to provide privacy protection for sensitive data without using cryptography.

This dissertation begins with a description a negative relational algebra and the complexity of each operation as applied to negative databases. Then, I propose how negative databases, managed by a negative database management system ($NDBMS$), can be used as an enabler for privacy-enhancing applications. In addition, I describe a way to represent negative data within a managed system and how to conduct negative database operations over them. I strive to develop an extensible architecture that can service both positive and negative data simultaneously. It is envisioned that having an automated system will encourage the adoption of negative databases for general database applications, especially in cases where database privacy is desired.

Privacy preserving applications, such as private matching, are the basis for several real-world applications. These applications enable different database owners to share information and get query results, such as set intersection, without revealing any other information to the other party. For example, with the increased emphasis on flight safety, airliners may want to verify that their list of passengers are not on the "no-fly" list of suspected terrorists. An airliner can privately match its flight manifest against the government's terrorist watchlist. However, for privacy reasons the airliner may not want to reveal its entire manifest containing other personal information to the government, and for security reasons, the government does not want to reveal its watchlist to airliners. Today, cryptological-based private matching protocols have been proposed to help solve these types of problems. In Chapter 4, I show how negative databases can also solve this problem.

On a related but different research thrust, Chapter 5 shows how to leverage the potentially compact size of a set's complement. Negative databases, using complementary information, to represent large positive sets, i.e., whose size is more than

half of its universe, are smaller since there are less elements to represent in its complement. This property leads to more compact storage and efficient algorithms for certain applications. One class of problems that may benefit from negative representation is that of variable tracking during set-sharing analysis of logic programs [61]. This application has the property that the positive data representation may yield an exponential number of relationships during execution. Set-sharing analysis determines which variables share, or dependent, and those variables (or set of variables) that are independent may be optimized for parallel scheduling or execution.

Current solutions, using positive representations, must contend with an exponential number of relationships. However, maintaining the negative representation of these relationships yields a smaller set size and enables more efficient algorithms for computing set-sharings. We explore this new paradigm proving that negative database concepts are applicable and beneficial to this class of problems.

We begin this dissertation with the essential background topics and previous work on this research area, i.e., negative databases, database systems, privacy-enhancing databases, and set-sharing analysis. Chapter 3 continues with a description of a negative relational algebra. A formal relational algebra as applied to negative databases is essential in furthering our understanding of how to better manage the negative representation of information. This helps explain which negative operations are efficient and which are not. Chapter 4 proposes an extensible negative database management system built over an existing relational database management system. Here we show a general architecture that can be used for managing large numbers of negative data among positive data. Chapter 5 shows how to use negative database concepts to increase the size of solvable set-sharing problem instances. Finally, this dissertation concludes with a summary and outlines possible avenues for future work.

Chapter 2

Background

2.1 Introduction

This chapter highlights previous works related to negative database concepts and applications. Negative database (NDB) ideas, as proposed by Esponda [44, 38], were inspired by how the immune system maintain and represents information. In particular, Esponda focused on the relationship of databases with the theory of self-nonself discrimination [63]. Self-nonself theory describes how the body's immune system detects antigens it has never before encountered and how it classifies these foreign substances for elimination.

Several immune system inspired algorithms have been proposed to help model this process, such as the clonal selection algorithm [35] and the negative selection algorithm [49]. The clonal selection algorithm emulates the way B-cells secrete antibodies killing the specific type of antigen detected. It also models how B-cells are stimulated by antigens to proliferate versions of itself to further kill encountered antigens. This model lends itself to evolutionary algorithms for simulation. On the other hand, the negative selection algorithm models how T-cells function. T-cells are

censored before distribution throughout the body so that only the ones receptive to cells other than itself (*nonself*) are allowed to leave the thymus [48]. Those that can bind to *self* are destroyed within the organ. Therefore, T-cells released to the rest of the body possess a high confidence that they are not harmful to the organism. These mature T-cells are able to bind and kill antigens never before encountered.

From an information representation perspective, one can think of a collection of all T-cells as a distributed model describing a set of data items *not* itself—its nonself. So, given a finite universe, if the collection precisely captures nonself, then it may also serve as a sufficient model to describe *self*.

2.2 Negative Database (NDB)

From these concepts, Esponda proceeded to map a representation of self and nonself as bit strings; thus, yielding a mechanism to transition from the immune system to database theory. More specifically, a positive set of binary strings representing self is consider the database (DB). For each positive database, the universe is based on a finite set of fixed length strings, created from a binary alphabet {0,1}. On the other hand, a negative database (NDB) is a representation of all elements not contained in a given DB.

A negative database compactly represents the complement of DB by introducing a special symbol, $*$, that allows a single NDB entry to denote many binary strings. A string with a $*$ at position i represents two strings, one with a 0 and one with 1 at position i (with all other bit positions equal). We define a negative database as a set of strings of length l defined over $\Sigma = \{0, 1, *\}$ that *match* (see Definition 5.4.2) each and every string in $U \setminus DB$, where U is the finite set of all possible binary strings of length l and $DB \subseteq U$. Thus, a negative database containing an entry with l $*$ symbols, $\{*^l\}$, represents 2^l binary strings. Table 2.1 illustrates a simple example.

DB	$U \setminus DB$	NDB
000	001	01*
101	010	11*
	011	0*1
	100	1*0
	110	
	111	

Table 2.1: A 3-bit example of a positive database (DB), all the strings not in DB, those in $U \setminus DB$, and its corresponding negative database (NDB) defined over $\Sigma = \{1, 0, *\}$.

Several algorithms for creating negative databases have been proposed, most of which are able to generate an NDB in time polynomial in the size of its input DB [44, 38, 109, 108]. The prefix-algorithm, for example, creates an NDB with at most $l|DB|$ entries in $O(l|DB|)$ time. However, the NDB produced by the prefix algorithm is easy-to-reverse and obtain the DB represented. In [45], Esponda et al. proposed a hard-to-reverse algorithm reviewed in the next section. Different negative pattern generation algorithms produce NDBs with varying properties depending on the number of $*$'s (the unspecified positions), its record length, and the total number of records in the database.

2.2.1 Hard-to-Reverse NDBs

Esponda et al. established a relationship between negative databases and Boolean expressions, specifically satisfiability (SAT) formulas [43, 44]. It was shown in [44] that the general problem of reversing a negative database to recover the corresponding DB is $\mathcal{NP}$-Hard using a reduction from 3SAT. In [43], they presented a new and efficient way to generate negative databases that are hard-to-reverse, i.e., it is extremely difficult to obtain the positive set from the negative representation. The algorithm takes advantage of the relationship of the negative data representation with

SAT formulas. In addition, they showed that by transforming a negative database into a Boolean formula and submitting it to state-of-the-art SAT solvers, an empirical assessment of an NDB's hardness is possible. In some cases, their method creates an inexact negative image of DB. In other words, other strings not in DB are also missing from the negative representation. This property further hides the true positive data among superfluous strings. However, in order to precisely answer queries, these extra strings must be distinguishable from the true data. They addressed this by including error detecting codes to precisely tag members of DB. They contend that in lieu of cryptography, hard-to-reverse NDBs may be used to secure sensitive data in applications where encryption is not desirable [43].

Hard-to-reverse NDBs were evaluated for their hardness empirically using two well-known SAT solvers, zChaff [9] and WalkSAT [87]. Esponda et al. showed that both solvers find a possible solution in time exponential to the length of the string [45]. In addition, candidate NDBs were deemed hard-to-reverse if neither solver was able to find a solution within 24-hours of computation or before running out of memory [43]. See [43, 37] for examples of how to generate hard instances. They consistently found negative databases to be hard-to-reverse when generated using records 1000-bits long and created using an algorithm proposed by Jia et al. in [64]. The algorithm was designed to hide a known satisfying assignment to a hard SAT instance. This hidden assignment is a member of DB represented by the hard-to-reverse NDB. However, a limitation of the process is that it hides only a single DB record. Thus, for each value in a database that we want to protect, we must produce a separate, *singleton* NDB ($SNDB$). In applications with storage limitations or those dealing with resource constraint hardware, hard-to-reverse NDBs may not be the best option. However, for applications where storage size is less critical, $SNDB$s may be a viable alternative to encryption.

As suggested above, not all instances of NDBs are hard in practice. More inter-

estingly, within the same application easy or hard NDB instances can be constructed depending on the generation algorithm used and they can interact with each other. Finally, situations may arise where DB is unknown and negative data is the only type of data available, e.g., remote sensors storing data that has *not* been sensed or when performing operations over existing $SNDB$s. In these cases, manipulating the negative database is the only option. Chapter 3 presents a relational algebra for negative databases enabling them to be treated as conventional relational databases.

Having many singleton NDBs in a database may pose a data management problem. As the number of data entries increase, keeping track of each negative database and its associations can be cumbersome. An automatic negative database management system would to help alleviate the administration burden associated with maintaining a large database. Chapter 4 proposes such a system.

Other security schemes based on $\mathcal{NP}$-Complete problems have been suggested, most notably the Merkle-Hellman cryptosystem [82] based on the general knapsack problem, but most of these schemes have been broken [93]. Other research efforts have looked into creating hard-to-solve SAT instances, e.g., [1, 26, 85, 64, 67, 102]. The flexibility of a relational algebra described in Chapter 3 could offset the lack of full cryptographic protection. This could be useful for applications in which relational operations over protected data is required, e.g., for limited searches and partial matching. Identifying such applications and deepening our understanding of the tradeoff between privacy protection and flexible data access is critical to expanding negative databases' utility.cannot read the data. And if the database is compromised, physically or logically, then the data will reveal nothing to the adversary.

2.2.2 Compressed NDBs

In this research, we also explore how NDBs can be used to exploit its compressed representation of the complement of large sets. Some applications may not require the security enhancing property of negative databases. Sets with a cardinality of more than 50 percent the size of its universe will have fewer elements in its complement. We explore whether the negative representation achieves favorable results in certain combinatorial problems. In particular, where input sets typically contain elements near its power set. Chapter 5 shows how NDBs can be used for such an application, called set-sharing analysis. In this chapter, a deterministic NDB generation algorithm is presented that attempts to produce the most compact NDB by reducing redundancies inherent in the ternary string representation. Associated operations to conduct set-sharing analysis using the negative representation are also defined. The ultimate goal of this part of the research is to increase the size of solvable instances of an intractable problem.

2.2.3 Negative Database Strengths and Weaknesses

Negative databases possess known properties, summarized below, that must be considered before using them. The following list briefly describes some of these properties:

• **Alternative to Encryption**. Computationally hard-to-reverse NDBs provide an alternative to encryption-based security algorithms [43]. Due to the similarities of encryption primitives used [36], alternative cryptographic primitives are desirable in the event one encryption system is broken, so that they are not all compromised. De Mare et al. proposed in [37] to exploit the hardness of finding solutions to SAT formulas and secure set membership problems as the building block for a secured protocol without encryption. Since NDBs maps directly to SAT formulas, NDBs

can also be used as primitives for secure set membership problems.

• **Partial Matching**. An essential property of good cryptographic protocols ensures that cipher text generated from two similar data (even those that differ by just one bit) are considerably different from one another. This offers great data security, but without prior knowledge of expected queries, partial data matching is impossible. Negative databases, by using $*$ symbols as wild cards, can perform partial matching over its obfuscated form.

• **No Key Management**. Key distribution and updates are management overheads that should not be ignored by any enterprise employing cryptographic tools. Negative databases do not use keys. They rely on the complexity of finding satisfying solutions, rather than some large numerical computation. Thus, no keys are required. This is especially beneficial when new users are granted access to shared data. They will not have to obtain the keys ahead of time. In addition, no periodic updates to keys are required.

• **Long-Term Storage**. Without keys, long-term data storage no longer depends on remembered keys or passwords. Since there are no keys to forget, lose, or compromise, a negative database remains available to service queries as long as its available.

• **Less Information Leakage**. Negative databases are suitable for privacy enhancing applications because when a portion of a negative database is compromised, the amount of information revealed by the negative is less than the positive database [38]. If an adversary acquires access to a portion of a positive database, then the adversary can inspect the records and know exactly which subset of records are in the database. On the other hand, if the adversary obtains and reverses a percentage of a negative database, any discovered positive records may still be contained in other uncompromised negative database. Thus, the adversary receives no guarantee that

captured records actually belong to the positive database.

• **Equivalent But Different**. Different versions of negative databases can represent the same positive database. Depending on parameter settings, i.e., number of specified bits, record length, random seeds, etc., a different negative database can be created representing the same positive database. As shown by Esponda, the Morph operation changes an existing negative database without actually changing its semantics [38]. This is convenient for preventing an adversary from deducing patterns within a database. After a set amount of queries, a negative database management system can automatically initiate a Morph operation and change the actual strings in a negative database without changing what they represent.

• **Closed Set Operations**. As shown in Chapter 3, set operations over negative databases are possible. These operations result in another negative database that represents a positive database as if the operations were conducted on the positive databases represented by the inputs.

• **Computation Time**. Basic operations, such as insertion and deletion of negative records, take longer, in comparison to a similar-sized positive database [38]. However, some relational operations over NDBs do take longer, while others are more efficient, see Table 3.4.

• **Hard Relational Operators**. Certain relational operators are $\mathcal{NP}$-Hard, namely Negative Project and Negative Set Difference. These operations presents unique challenges that must be addressed. These operations should be avoided as much as possible. For special cases, these operations can be compensated for, e.g., designing the database schema around Negative Project, see Figure 4.3.

• **Smaller Complement Set**. Negative databases that represent large positive sets, i.e., more than half of its universe, result in smaller sets. Less storage and more efficient computations are possible for certain applications, see Chapter 5.

• **Large Databases**. On the other hand, for small positive databases, negative databases are usually much larger in size. This size difference may not be suitable for applications with memory constraints or without any security requirements. At this time, only single record DB can be converted into hard negative databases. These hard-to-reverse, singleton NDBs are even larger than other types of NDBs.

• **New Paradigm**. A paradigm shift is required in developing algorithms for performing straight-forward positive database operations. What seems like a simple operation in positive databases must be carefully examined when working with its negative representation. For example, see how the negative relational operations (Chapter 3) and the negative abstract unification operations (Chapter 5) are defined.

2.3 Databases

A database is a structured collection of data for use by a variety of users and systems [103]. For this research, our data is defined as a set of binary or ternary strings of a given length. This set of bit strings can be used to represent a specific value or character of interest. For example, an ASCII character may be represented by its 8-bit binary value. Combining multiple bytes, we are able to store textual information up to the given record length. In addition, in Chapter 5 we use bit strings to represent characteristics of a given problem instance. For example, in the set-sharing analysis, a 1 in position i of a string signifies that a particular variable, represented by the i^{th} bit, is present in a specific sharing group.

Within a database, there commonly exists a consistent and organized way to represent the underlying data. This data model is comprised of a collection of tools for describing and manipulating data, relationships, and semantics. A defined data model helps organize and visualize the data from various users' perspective. In [103], several data models are described, including:

• Relational Model. A collection of tables represents both data and the relationships among those data. Each table has multiple, uniquely named columns corresponding to attributes of specific record types. It is the most common data model.

• Entity-Relationship (E-R) Model. Consisting of a collection of basic objects (entities) and their relationships based on the view of the real world.

• Object-Based Data Model. An extension of the E-R model with notions of encapsulation, inheritance, functions, and object identity. The object-relational data model combines features both the object-based and relational model. Most modern database system incorporates some form of data object handling.

• Semistructured Data Model. Specifies, normally by tagging, data items of the same type to have different sets of attributes. This model works well for specifying web pages and extensible markup language (XML) documents.

The data model determines how the data are viewed, organized and accessed. In addition, the model determines the definition and manipulation language used to query the database. Our research in negative relational algebra and database management system uses the relational model as the negative data model.

In a relational model, users (or programs) interact with a relational database by sending it a query written in some version of Structured Query Language (SQL). In response to the query, another relation consisting of records (or rows) constituting the answer is returned. The simplest query returns all the rows from a given table, but more often, the rows are filtered and/or combined in some way to return just the desired answer. Relational operators are used to manipulate the relations accordingly. The minimally complete set of relational operators, its relational algebra, consists of: select, project, union, intersection, set difference, and Cartesian product. We define a complete relational algebra for negative databases in Chapter 3.

2.3.1 Database Management System

Databases have been around for many years. They are a component of a wide range of applications, from an enterprise level information system to basic single-user catalogs. To manage a database more efficiently, a database management system ($DBMS$) is usually employed. A $DBMS$ is a set of programs used to organize and access a collection of inter-related stored data [103]. In 1972, the American National Standards Institute (ANSI) proposed an architectural framework for databases that has been widely adopted [32]. The standard proposed three levels of abstraction: the external, conceptual, and internal views. The internal view is the one closest to physical storage and how the actual data are laid out. The external view is closest to the user; thus, it is concerned with how the data are viewed by individual users, and it determines how users will interact with the database. The conceptual view acts as the intermediary between the internal and external views. The conceptual view is concerned about the database schema that is viewable by the entire community of users. In this way, each level coincides with three different views (external, conceptual, interna) of the data, the individual user view, the community user view, and the physical view, respectively [103]. In addition, the different levels provide a separation of information semantics from level to level. This adaptable architecture provides the flexibility and data independence needed to develop a negative database management system ($NDBMS$), see Chapter 4.

Current negative database implementations rely on flat files as the data organization for persistent storage. This research implements the relational operators using a $DBMS$-based system. A system perspective provides several advantages. Centralized data management can take advantage of availability of all data from a single view. This may reduce redundancies and inconsistencies of data. For example, a person's address may be stored in multiple location (redundant) and whenever an update occurs, all copies must be updated leading to a potential for inconsistencies.

Storing data in a central $DBMS$ allows designers to better establish and enforce standards for data format, naming conventions, and database schema. Another advantage of a $DBMS$ is it promotes data independence from applications using the data.

On the other hand, a centralized database managed by a $DBMS$ has several disadvantages. For smaller, customized applications, a $DBMS$ is usually not necessary. It adds complexity and increases the size of the program. In addition, maintaining data in a central location introduces a single point of failure for the system. More importantly, since all data are now stored in one location, without robust access control, data may be revealed inadvertently to unauthorized users. Private information may be leaked by the $DBMS$ to users only authorized access to portions of the entire database [32, 51]. Data owners should be able to select the level of protection for each data attribute at the internal view.

This research builds on a popular open-source $DMBS$ called PostgreSQL. It is an object-relational $DBMS$ licensed under the Berkeley Software Distribution (BSD) agreement. It has had many years of active development and a proven architecture earning it a strong reputation for reliability, data integrity and correctness [94]. Building over a robust system relieves us from reimplementing many common algorithms, i.e., matching, sorting, etc.

2.4 Constraint Databases

A related scheme for compactly representing large datasets in a relational format is called constraint databases. They generalize relational databases by including associated constraints in the query and/or within the data model itself [7, 96]. These constraints normally take the form of linear or polynomial equations. They have been used extensively to characterize infinite sets [65]. Their ability to deal with

infinite sets makes them particularly promising as a technology for integrating spatial and temporal data with relational databases. It may be possible to represent specific negative data, with known semantics, using constraint database constructs. However, in general, it is unclear how the negative representation independent of the semantics of the data can be represented correctly by any equation. In addition, the data will no longer be secured since the equation will reveal the semantics of the representation. Although negative databases use the relational model, constraint databases operate on the logical level for a compact representation while negative databases operate more on the internal representation, i.e., at the bit level. In addition, unlike typical constraint database operations, negative databases represent finite sets using a straightforward data model (without explicit constraints). Negative databases organically induce constraints based upon its representation and the types of operations that can be performed efficiently over them.

2.5 Data Privacy

Today's technology and communication infrastructure provides unlimited access to vast amounts of data. This two-edged sword has privacy advocates clamoring for more protection of private information. In addition, the trend towards outsourcing database services has further fueled concerns [98, 111]. Furthermore, as a response to heightened privacy concerns and identity theft, recent laws mandate companies to notify customers and the public of incidents that potentially expose their private data [98, 111], along with other information security laws and regulations designed to protect the privacy of personal information [71, 72, 73, 74, 75, 76]. Such financial exposure by companies may lead to loss of revenue from lawsuits, civil fines, and/or consumer confidence. These changes to the business environment are forcing industry to rethink its approach to database management and privacy security.

In this dissertation, I show how negative databases may be used to ensure data privacy by leveraging both its positive and negative representations. In addition, I show that negative databases can be treated like ordinary relational databases.

2.5.1 Cryptographic Schemes

Cryptographic techniques are the most common solution to data privacy. Thus, they play a major role in maintaining data confidentiality in current database systems. Even though the encryption/decryption process can be computationally expensive, it is one of the most secure and reliable techniques for keeping databases private [79]. Another benefit of encryption is that it also can guarantee data integrity. Any unauthorized alterations can be recognized, i.e., using a hash signature. Hence, an investigation can occur and data can be reconstructed from a backup copy [28]. It is obvious that this technique meets the requirement of keeping databases private. However, the computational expense of cryptography and the associated key management overhead may not be appropriate for all types of applications. Also, as stated earlier, once encrypted, partial matching is prevented. In other words, encryption is a binary all or nothing form of protection.

Several schemes exist to combine cryptography with databases. In 1981, Davida et al. evaluated several of these cryptographic techniques, i.e. public key encryption and symmetric encryption at table, record and field levels. More importantly, he described characteristics that database encryption systems should possess to ensure a practical database management system [34]. They include the following:

• Encryption system does not need to be theoretically secure, which is impossible to guarantee. Systems based on classically hard problems are sufficient.
• Encryption and decryption must not degrade performance to unacceptable levels, especially decryption, which is performed more often during query execution.

• Encrypted data must not have significantly greater storage volume than unencrypted data.
• Encryption must occur at the record level of a database. Since the position of a record is likely to change during its lifetime, there should be no need to decrypt $n-1$ records to get to the n^{th} record.
• Encryption should support logical view presentation depending on different user privileges.
• An encrypted record must not contain a series of individually encrypted fields. Such a scheme can lead to pattern matching and substitution attacks.
• Reads and writes to the database must be allowed without any special constraints.
• The systems should be able to detect and reject data using false encryption key, even without knowing what the data contain.

In Chapter 4, we address each of the above characteristics as they relate to negative databases. Davida recommends that the encryption system be record-oriented where a single record is a single function of all its field values. Additionally, each field is encrypted/decrypted by means of a separate key. He called this scheme an encryption system using subkeys [34]. Subkeys have many advantages over either table or block level encryption. It provides better granular control preventing pattern matching attacks and the possibility of substitution of fields. To enhance performance, specific records (and fields) can be decrypted without decrypting the entire block. This property is ideal for databases, where records are usually aggregated to answer a query; however, not all fields should be visible to everyone.

A major disadvantage of this scheme is that updates are expensive. The entire record must be re-encrypted then rewritten, which can severely degrade performance if there are frequent updates to the database. Another possible drawback is when the database is visible to an outsourced database management service provider; it can

reveal private information. System manipulation by extracting encryption keys or saving queries and results to try to crack the customer's keys or use pattern matching to gain private information, may be possible.

2.5.2 Encryption with an untrusted service provider

A more advance approach models database transactions with a trusted client and an untrusted, database management service provider [28, 57]. Conventional approaches to database encryption protect the data in storage and assume trust in the server. The server decrypts the data for query execution then re-encrypts them before transmission and subsequent storage.

In 2003, Damiani et al. proposed, an extension of work done by Haciqumus [57], where the trusted client side performs a mapping of plain text queries to new queries based on the encrypted values stored in the untrusted database service provider [28]. In this model, in order to keep data at the provider's end encrypted, new indexing information is needed. The new index is based on the encrypted data value, which can then be used to answer queries based on ciphered data. The query result is sent to the client for post-processing to readable text.

Their technique assumes that encryption occurs at the record level. However, both researchers were challenged with the best way to index encrypted data. They sought a good way to create and represent indexing information. Taking privacy protection into account, they note two conflicting requirements for a suitable indexing scheme. First, the indexing information should be close enough to the data to expedite query execution and optimization. On the other hand, the indexing information should be obscure enough to prevent inference and pattern matching attacks that will compromise the confidentiality of the database [28].

They recommended indexing over encrypted data rather than the unencrypted

data or metadata. A record can be encrypted using a suitable encryption scheme or a secure hash function over the attribute values. Then, an index is created over ciphered texts. They also defined a metric for unwanted inference, which they called the inference exposure coefficient [28]. Basically, it measures how close the encrypted data is to the plain text data.

Other techniques for privacy preserving set operations, for example private set intersection, have been proposed, all using cryptographic schemes. The following introduces several related topics.

2.5.3 Secure Multiparty Computation

Secure Multiparty Computation is a protocol that allows a group of people to compute any function, based on individual confidential inputs. The result is known to everyone but no one learns anything about the inputs of any other members [100]. This problem is sometimes known as Yao's millionaire problem [112], where each millionaire wants to know who is the richest of the group without revealing their actual wealth to others. It is based on a protocol that uses homomorphic encryption scheme [100]. Secure Multiparty Computation provides the foundation for solutions to many real-world problems such as distributed voting, private auctions, secure signature sharing and others covered below.

2.5.4 Private Matching

Private matching is a special case of Secure Multiparty Computation. It has been called by other names, such as private selective data sharing or private set intersection problem [3, 69]. Two different data owners would like to collaborate and find common data between their databases. However, for the protection of the privacy of its data,

neither one of them wants to openly share all of their data nor do they want to reveal their query parameters to the other party. This problem can be illustrated by considering the case of an airliner and the Federal Bureau of Investigation's (FBI) watchlist of suspected terrorists. The airliner would like to query the watchlist database but also wants maintain its customers' privacy. So, it would rather not reveal the names and other personal information of its customers as they query for a match against the watchlist. The FBI would also rather not reveal all the people on its watchlist for security reasons. Using private matching protocols, both the airliner and FBI can share common data, the intersection of names or profiles, with each other without revealing additional information.

Current solutions to private matching rely either on a trusted third party or cryptography to ensure the privacy between the data client and server. A trusted third party maintains a copy of all databases needed. Then, it computes the desired set operations on both databases. The result is then returned to the query provider. Neither the query provided nor data owner learns anything about the other party's query or non-relevant data. However, this solution completely relies on the third party for protection against misuse and security breaches. In most cases, the level of trust required is too high for this solution to be acceptable.

Cryptographic solutions to private matching employ homomorphic encryption [50, 3, 69] along with a straightforward communication and matching protocol. Each party encrypts its data, the query provider sends its query to the other party, then the server conducts a search for matching encrypted records. Those that match are sent to the query provider where the subset is decrypted. The operations that have been supported include set-intersection, cardinality of set intersection, and set union. This capability is similar to our NDB relational algebra operations for query servicing.

2.5.5 Private Information Retrieval

Private information retrieval schemes hide the values retrieved by queries from the database server. This protocol enables a user to query the i^{th} bit of an n-bit database or document, while keeping the value at position i private [52] without transmitting the entire database. To achieve this goal, the user queries multiple, non-cooperating servers and the replies received are compared and used to compute the value at position i.

Private information retrieval using distributed databases was introduced in 1995 by Chor et al. They showed that it was possible to efficiently use replicated, but segregated databases to conduct private information retrieval [18]. With two databases, they showed that their protocol can achieve sub-linear communication complexity. At that time, no one was concerned about the privacy of the user (query provider) of the database. However, it was later deemed appropriate to consider the query provider's privacy as well. For example, a stock investor may be interested in specific stock information but wants to keep private which specific stocks he is currently researching. One way to perform this query is to ask for the entire database and to conduct the queries at the user site. However, this may be impractical for large databases due to communication complexity, or database owners may not be willing to provide their entire database. In 2003, Gertner et al. extended Chor's solution and developed the Symmetrical Private Information Retrieval model guaranteeing both user and data server privacy [52]. So, a user's query is kept private from the server, and of equal importance, the user only learns the result of his query and no other information from a single query. With this solution, private information retrieval becomes closely related to private matching (or private set intersection) in that one can think of private matching as multiple rounds of the private information retrieval protocol over a single database.

2.5.6 Searching Over Encrypted Data

Searching over encrypted data and retrieving documents containing specific words is difficult, especially without loss of data confidentiality [105]. A client can either submit a plain text query to a trusted server, then the server encrypts the text for matching. In this case the server learns which values the client is interested in, which may violate the privacy of the client. Another scheme assumes an untrusted server and the client creates the encrypted database for future use.

In [105, 106], Song et al. proposed Searchable Symmetric Key Encryption (SSKE) scheme, introducing a way to search encrypted documents in linear time. The general word-based approach introduced by Song et al. provided keyword-based schemes that were used by Boneh et al. in [8] using public key encryption. In [11], Brinkman et al. developed a more efficient tree search algorithm based on the linear search algorithm proposed by Song et al. that is suitable for semi-structured databases and documents.

In [54], Goh introduced the concept of a secure index, which allows the user to search for a word in the index if and only if the user has a valid match with an encrypted index entry. The index does not reveal any other information about its contents to the requester. Their schemes returns which documents contain the keyword provided. It assumes that each client has access to the same key or set of keys used to generate the psuedorandom strings used to encrypt the document. This may not be practical in a multiple user environment.

In [56, 57], Hacigumus et al. proposed an architecture that supports performing SQL queries over encrypted data. Their technique employs an index, which allows partial execution of an SQL query on the server side. The result of this query is sent to the client. The correct result of the query is found by decrypting the data, and executing a secondary query at the client site. Each of the above protocols perform

some form of cipher text matching to find the desired records. However, unlike negative databases, they cannot perform partial matching within the obfuscated text.

2.5.7 K-anonymity

One related research area that has gain attention using a non-cryptographic approach to share data privately is the process called k-anonymization. K-anonymizaty provides information, to trusted or untrusted clients, while preventing unwanted release of private information. A piece of information is considered releasable if it cannot be distinguished from k other records in the query result. As an example, hospitals may collaborate in order to catch an outbreak of an epidemic. This collaboration requires access to patients' medical records containing sensitive information. In this case, data should be provided to allow researchers to draw inferences without violating the individual's privacy [2]. More specifically, a query result cannot lead to identifying specific attributes belonging to a specific person.

One naive way of preventing identification is to remove explicit identifiers from the database tables, e.g., filter out names, addresses or patient numbers. However, it has been shown that even when explicit identifiers are removed from the database table, non-identifying attributes can lead to narrowing and inferring specific individuals. It has been shown that it remains possible to discover who has what disease using a public database and voters' list, even when identifying attributes, i.e., names and social security numbers, have been removed [107].

As a rudimentary example, Table 2.2 contains patient health information used by researchers for a medical study [114]. Each row consists of a patient's date of birth, zipcode, allergy, and history of illness. Note that no patient identifier explicitly appears along with the table. However, a crafty adversary may be able to deduce the

DateofBirth	*Zipcode*	*Allergy*	*HistoryofIllness*
10/7/69	87111	Penicillin	Stroke
8/29/65	87113	None	Polio
9/22/66	87110	None	Colitis
8/29/65	87119	Sulfur	Diptheria
4/16/68	87113	None	Stroke

Table 2.2: Patient Health Data

identity of patients using date of birth and zipcode, i.e. by using another information source such as the voters or driver's license database.

In the above example, the subset of attributes that may lead to an identification, thus release of private information, is called a quasi-identifier. In this case, one quasi-identifier is the set {date of birth, zipcode}, which uniquely identifies an individual with significant probability. Other attributes such as allergy and history of illness are sensitive attributes, while others are deemed irrelevant.

A breach of privacy occurs when an adversary can link the sensitive attributes to corresponding individuals using information provided by the quasi-identifiers. The process of k-anonymizing a database ensures that the quasi-identifiers appear at least k times in any table. Therefore, if an adversary wanted to use the query result to link sensitive attributes to individual identifiers, then each entity is masked by at least k peers [114].

This is accomplished by eliminating unique attributes using a new character, i.e., a $*$, to partly or entirely obscure the data. Table 2.3 shows the resulting 2-anonymous table of the previous table.

Notice that the count value of the quasi-identifier appears at least 2-times in the table. In a more general case (larger k value), an adversary will not have high confidence by linking quasi-identifiers to identify individuals. This process can be

$Date of Birth$	$Zipcode$	$Allergy$	$History of Illness$
*	8711*	Penicillin	Stroke
8/29/65	87113	None	Polio
*	8711*	None	Colitis
8/29/65	8711*	Sulfur	Diptheria
*	87113	None	Stroke

Table 2.3: 2-Anonymized Patient Health Data from Table 2.2

achieved by either suppression or generalization. Suppression replaces some entries with a new character. So, the three unique date of births were replaced with a $*$. Generalization replaces characters within each field. For example, 87111, 87110, and 87119 generalize to 8711*.

It is important to note that this obfuscation does not use cryptography in the process. In most cases, it is desirable to provide as much information as possible while maintaining the desired level of privacy. However, in general this balance is difficult to accomplish. Meyerson and Williams have shown that minimizing the number of suppressed entries in k-anonymization is $\mathcal{NP}$-Hard [83]. They provided approximation algorithms for the problem running $O(klogk)$ with a constant of no more than 4. A survey paper by Ciriani et al. shows the various complexity for several k-anonymization algorithms [19].

Although $\mathcal{NP}$-Hard in general, many researchers believe that k-anonymity is quite fast in practice [3, 19, 99, 107]. For more static databases, potentially better performance and higher gains can be achieved in the area of private information sharing.

2.5.8 Data Hiding

A more recent research paper was published by Esponda in the area of data hiding [40]. It uses negative database as a tool to obfuscate the original data within a larger set supplemented with similar but meaningless records. These extra records act as "chaff" obfuscating the true data amidst many other strings, making it difficult to systematically differentiate valid datum from superfluous answers.

This technique takes advantage of the numerous superfluous data records not captured during a new negative database generation process. By using this algorithm, the size of the negative database need not be as large as the hard-to-reverse, singleton NDBs [45]. The original data is secured by relying on the number of superfluous entries and the infeasibility of retrieving only the valid items. Similar to the original hard-to-reverse protocol, valid strings are augmented with a code based on the value of the original string. In this way, membership queries remain efficient with respect to the size of the database. A new contribution of this scheme is that valid strings are effectively hidden, among many invalid strings without relying on the $\mathcal{NP}$-Hardness property of negative databases, i.e., hard-to-reverse $SNDB$s.

2.5.9 Security Model

Typical adversarial models used to evaluate the feasibility of privacy preserving applications are the *malicious* and *honest-but-curious* models [3, 69]. The malicious model assumes that an adversary will do whatever is needed to obtain private information from the other parties. Any and all types of vulnerability attacks are at this adversary's disposal. This research did not consider attacks outside of direct manipulation of the negative database. The main focus of this research is on leveraging both positive and negative representation to achieve data privacy and improve efficiency. We employ hard-to-reverse $SNDB$s as proposed by Esponda et al. in [45].

This research did not consider the malicious adversarial model, but it assumed that two parties that share data are honest-but-curious. In other words, the requester and data server follow the given protocol correctly, but they may hold on to intermediate results to gain some other information regarding the client's database. We also assume that the server will act appropriately, i.e., will not withhold or change query results. Additionally, this research did not consider a trusted third party solution which gives unconditional custody of ones data.

Data encryption at storage and transmission is the most common solution to ensure database confidentiality and privacy. In some situations, encrypting an entire database may significantly slow down performance due to queries requiring encryption or decryption. A loosening of this restriction, only to protecting sensitive data at field level, is sufficient. We take this approach in the design of the $NDBMS$. An entire record need not be in the negative format but only those fields within a record deemed sensitive.

Considering the tradeoffs discussed in Section 2.2.3, negative databases may be a suitable surrogate for protecting sensitive data currently reserved for cryptographic tools. Sharing of positive records is convenient but not desirable from a privacy point of view. On the other hand, an NDB may not be as efficient, but it does reveal little information directly. This characteristic is desirable especially when the multiple databases are stored in different, semi-trusted locations. For this technique to be implemented, the NDB must be manageable and provide similar mechanisms used by traditional $DBMS$. This research area is rich in potential and is explored in Chapter 4.

2.6 Set-Sharing Analysis

A promising research area is the potential for exploiting the compact negative (complementary) representation of large sets to alleviate computational efficiencies when solving combinatorial problems. A major component of this dissertation shows how negative database concepts can be applied toward such an application.

Abstract interpretation is a process of generating conservative approximations of the semantics of a program. It can be used to determine run-time properties by analyzing the code statically. Knowing the properties of a program's run-time behavior can be useful for debugging, code optimization, program transformation and proof of program correctness [27].

One form of abstract interpretation as applied to logic programs is called set-sharing analysis. Two or more variables in a logic program are said to *share* if, in some execution of the program, they are bound to terms that contain a common variable. A variable in a logic program is said to be *ground* if it is bound to a term that does not contain free variables. *Set-Sharing* is an important type of combined sharing and groundness analysis. It was originally introduced by Jacobs and Langen [61, 70] and its abstract values are represented as sets of sets of variables that keep track of sharing patterns among variables.

However, due to the large number of variable combinations that must be evaluated during set-sharing analysis, an intractable number of sharings may occur. Due to memory constraints and/or long computation time during analysis, compromises are commonly made between precision of the analysis and its tractability. Therefore, any steps toward a more compact set-sharing representation along with efficient, yet precise, operations are desirable to further increase the size of solvable set-sharing problem instances. We selected the area of set-sharing analysis partly due to a collaborative effort with a programming language research team at our department.

Negative database concepts have enabled us to increase the size of solvable set-sharing problems. Chapter 5 covers our extensive research in this area.

2.7 Summary

This chapter presented previous work used as background material for extending the research on negative databases. These topics help further establish the foundation for negative database critical to applications presented in later chapters. We began by presenting what negative databases are and highlighted several publications with regards to this new representation. Then, a review of database and database systems was provided along with criteria to help guide the design of our negative database management system. It is from these ideas that we are able to integrate negative databases into an ordinary relational database management system.

Previous research has shown that certain NDBs can be hard-to-reverse and may provide a viable alternative to data privacy without encryption. Next, I reviewed privacy-enhancing techniques for use with databases (with most resorting to cryptographic techniques). Lastly, I introduced the area of set-sharing analysis, which gives us an opportunity to use NDBs as a compact representation of very large sets. Each chapter of this dissertation will cover other specifically related work in greater detail as needed.

Chapter 3

Relational Algebra for Negative Database

3.1 Introduction

This chapter presents a relational algebra for negative databases. The goals of this part of my research include gaining a better understanding of how a formal algebra can be applied to negative databases and studying the complexity of each operation in comparison to its positive counterpart. A large part of this chapter is excerpted from a technical report [46] written in collaboration with F. Esponda, with E.S. Ackley contributing. Esponda and I initially developed the beginnings of a negative relational algebra separately. Then, we decided to collaborate and merge our efforts. The algorithms and notations used here evolved from an initial draft written by Esponda. They have since been refined several times for correctness and clarity.

This chapter defines the minimal set of relational operators (select, project, union, intersection, Cartesian product, and set difference). As a consequence, we are able to establish a better relationship between negative databases and positive relational

database theory. This insight provides us with an improved understanding of how to organize negative data and the types of queries efficiently supportable by negative databases. This work advances the practicality of negative databases and expands their range of application.

As introduced by Esponda in [38], the initial negative database framework supported only membership queries and operations for inserting to and deleting from a negative database. This chapter extends that work by defining a set of relational operators for the negative representation. In the following sections, a negative operator is defined such that the result (another negative database) of the negative operator applied to a negative representation is equivalent to the positive operator applied to the positive representation. In other words, each relational operator takes negative databases as input and returns another negative database representing what would have been the positive result.

Sets are one of the most fundamental mathematical constructs and are, consequently, pervasive throughout computer science. In particular, data collections can be viewed as sets, and operations on sets are translated into operations on databases. A relational algebra, introduced by E.F. Codd in [20], treats databases as sets and defines operators that form the foundation for many database management systems.

Traditional databases typically store data designed for ease of access and manipulation. However, data privacy and security issues are usually an afterthought and are implemented on top of existing database architecture. Our approach is to embed data obfuscation organically within the data representation itself by storing the complement of the set of interest using a ternary alphabet. So, instead of storing the records of interest explicitly, all the records not in the original database are stored. This alternate representation is known as a *negative database* (NDB). Algorithms for creating and storing NDBs efficiently are given in [37, 43, 44, 38, 108, 109].

One motivation for a negative database is to restrict how easily information about the original set can be exploited, even in the case of insider threat. The positive data represented can be difficult to retrieve without resorting to enumerating all possible elements. This privacy enhancing aspect of negative databases is achieved as a result of the relation between some negative data representations and Boolean satisfiability formulas [43]. Furthermore, the operations defined in this chapter support this paradigm enabling negative databases to be manipulated without specific knowledge of their contents.

For example, in a database containing names, social security numbers (SSN) and bank account numbers, an operation can be defined to compute negatively only the subset of DB that has SSN records from a region of the United States—the first three digits of American SSNs show the geographical region where the number was issued. Then, another negative database of suspected terrorists' bank account numbers can be *joined* to this restricted result using the bank account number field to produce a third NDB for a watchlist in the negative format. Several other NDBs can be combined without explicitly learning the positive sets, affording privacy to other Americans with SSNs from the same geographical region.

In addition to the data hiding aspects of negative databases, there is the potential for exploiting negative representations to achieve computational efficiencies. As Section 3.2 shows, some simple operations using positive databases become hard (this is the basis for the privacy enhancing features), and some difficult operations are simplified. For example, by applying de Morgan's law, the Negative Intersection of two databases can be computed using negative databases by simply concatenating two sets to form a Union (see Section 3.2.6). Section 3.4 describes how the complexity of relational operations can be harnessed to support program verification.

This chapter is organized by first defining the relational operators over negative databases giving algorithms, proofs and time complexities (Section 3.2). An example

scenario (Section 3.3) is provided along with an implementation. In addition, how the operators can be applied to problems, their limitations, and strategies addressing these issues along with empirical results (Section 3.4) are presented. Lastly, a review of related work (Section 3.5) and a summary of the important points are provided (Section 3.6).

3.2 Relational Operations

This section defines a series of operations on sets that correspond to the well-known relational algebra operators select, union, Cartesian product (along with join and set intersection), project, and set difference. Each operation takes one or more NDBs as input and produces a new NDB representing the strings that are *not* in the result of applying a traditional relational operator to positive databases. Throughout this chapter we consider a set of unique ternary bit strings as the database. Thus, the terms *sets* and *databases* are used synonymously along with *strings* and *records* as elements in a set or database, interchangeably.

First, we introduce the following notations and definitions used throughout the chapter:

- $\Sigma = \{0, 1, *\}$: a ternary alphabet.
- Σ^l: finite set of all strings over Σ with length l.
- x, y, z: strings, a concatenation of symbols from Σ^l.
- $x[i, ..., j]$: string x projected onto positions $i, ..., j$.
- b^m: string consisting of character $b \in \Sigma$ repeated m times.
- xy: string x followed by string y (concatenation).
- U_m: the universe of all possible binary strings of length m; if $m = l$, written as U.
- Ω_m: an ordered list of all m positions; if $m = l$, simply written as Ω.
- Υ: an ordered list of string positions of interest.

- $[\Upsilon]$: bit values at positions specified by Υ.
- DB: a positive database, $DB \subseteq U$.
- NDB: a negative database representing $U \setminus DB$. Note: $U = DB \cup NDB$.

The following definition is used to determine if two ternary strings represent a common string or set of strings.

Definition 3.2.1 (Match, $\mathcal{M}$). Given two ternary strings of length l, $x, y \in \Sigma^l$, *match* is a function $\mathcal{M} : \Sigma^l \times \Sigma^l \rightarrow$ `Boolean`, such that $\forall i\ 1 \leq i \leq l$,

$$x\ \mathcal{M}\ y = \begin{cases} \texttt{true}, \text{if } \ (x[i] = y[i]) \vee (x[i] = *) \vee (y[i] = *) \\ \texttt{false}, \text{otherwise} \end{cases}$$

Therefore, when two strings x and y *match*, it means that they represent some common string(s), i.e., they have a non-empty bitwise intersection. However, x and y may represent other strings not in their intersection. This leads to an important property that two sets of ternary strings may represent equivalent positive databases without containing the same exact records. In other words, two databases can be *equivalent* without being *equal*. For example, if $NDB_1 = \{1*1,\ *11\}$ and $NDB_2 = \{1*1,\ 011\}$, then $NDB_1 \equiv NDB_2$ even though $NDB_1 \neq NDB_2$ (they contain different strings).

One possible internal representation of a negative set is to treat it as a table where each column corresponds to a single character, {0,1,∗}, see Table 3.1. This model is discussed further in Chapter 4. We design the model to facilitate the use of Structured Query Language (SQL) queries, as illustrated below. Given a set of ternary strings, a database can be constructed using a character for each bit position. So, for each ternary string in the database, there is an l-bit record representing it. Even within a relational data model, there are other possible representations; we

$Tablename_1$	
b_1	CHAR(1)
b_2	CHAR(1)
.	
.	
.	
b_l	CHAR(1)

Table 3.1: Sample SQL Table for Ternary Strings

present the most straightforward one here. An alternative representation is described in the Appendix.

Given this model, we can service membership queries of a candidate string using SQL. First, the query string must be saved as a record in a database table, say $QueryDB$. Assuming our negative database is called $NDB1$, the following SQL query can be constructed to check the membership of $x \in DB$ represented by $NDB1$.

```
SELECT true FROM QueryDB qdb
   WHERE EXISTS (SELECT true FROM NDB1 ndb1
      WHERE (qdb.b1 = ndb1.b1 OR qdb.b1 = '*' OR ndb1.b1 = '*')
        AND (qdb.b2 = ndb1.b2 OR qdb.b2 = '*' OR ndb1.b2 = '*')
        AND ...
        AND (qdb.bl = ndb1.bl OR qdb.bl = '*' OR ndb1.bl = '*'));
```

A function can be used to dynamically create the above query given the argument. In addition, certain error checking can be accomplished ensuring that the query string is of equal length to the database entries. Other relational operations presented below can be created using a procedural language compatible with a database management system, e.g., PL/SQL, C, Java, PHP scripts, etc.

Throughout the following discussions, refer to Table 3.2 for examples of each relational operator for positive data and Table 3.3 for examples. For each operation,

DB_1	DB_2	$\sigma_{[\Upsilon]=101}(DB_1)$	$\times$	$\bowtie$	$\cap$	$\cup$	$\pi_{\Upsilon=1,2}(DB_2)$	$DB_1 \setminus DB_2$
001	001	101	001001	0010	001	001	00	101
101	010		001010	1010		010	01	
			101001			101		
			101010					

Table 3.2: Two positive databases with 3-bit strings, and the result of applying the indicated operations (select, Cartesian product, join, intersection, union, project, and set difference). Note the join condition for $\bowtie$ are positions $\Upsilon_1 = \{2, 3\}$, $\Upsilon_2 = \{1, 2\}$.

NDB_1	NDB_2	$\overline{\sigma}_{[\Upsilon]=101}(NDB_1)$	$\overline{\times}$	$\overline{\bowtie}$	$\overline{\cap}$	$\overline{\cup}$	$\overline{\pi}_{\Upsilon=1,2}(NDB_2)$	$NDB_1 \overline{\setminus} NDB_2$
01*	000	01*	01****	01**	01*	011	10	01*
*00	011	*00	*00***	*00*	*00	000	11	*00
11*	10*	11*	11****	11**	11*	100		11*
	11*	0**	***000	*000	000	11*		001
		1	***011	*011	011			
			***10*	*10*	10*			
			***11*	*11*	11*			

Table 3.3: The results of relational operators on negative databases, NDB_1 and NDB_2. Corresponding results complement those from Table 3.2. Note: the * symbol stands for both 0 and 1 and the join condition for $\overline{\bowtie}$ are positions $\Upsilon_1 = \{2, 3\}$, $\Upsilon_2 = \{1, 2\}$.

its corresponding time complexity is presented in Table 3.4.

3.2.1 Negative Select

The Select operation over a positive database restricts the relation according to some criterion in the form of a predicate. Select is defined in terms of a relation between two attributes (here understood as the values at some string positions), or an attribute and a constant v. We consider the basic binary operators, $\theta = \{<, \leq, =, \geq, >\}$. We limit our description to the latter case, leaving the former for future work. Thus, the Select operation applied to DB is defined as:

$$\sigma_{[\Upsilon]\theta v}(DB) = \{x \mid x \in DB\} \ \cap \ \{x \mid x[\Upsilon] \ \theta \ v\}$$

where Υ is an ordered list of string positions. The complement of this set is written as:

$$U \setminus \sigma_{[\Upsilon]\theta v}(DB) = \{x \mid x \notin DB\} \cup \{x \mid x[\Upsilon] \ \overline{\theta} \ v\}$$

where $\overline{\theta}$ stands for the opposite relation of θ, i.e., $\overline{<}$ stands for $\geq$, $\equiv$ stands for $\neq$, etc. In the following, we describe how to implement $U \setminus \sigma_{[\Upsilon]\theta v}(DB)$ using negative databases.

Let $NDB_1 = U \setminus DB$, and let NDB_2 contain all the strings in U that do not satisfy the criterion, i.e., $\{x \mid x[\Upsilon] \ \overline{\theta} \ v, x \in U\}$. Negative Select is defined as:

$$\overline{\sigma}_{[\Upsilon]\theta v}(NDB) = \{x \mid x \in NDB_1\} \ \overline{\cap} \ \{x \mid x \in NDB_2\}$$

The above formula shows that Negative Select can be viewed as the Negative Intersection, which is equivalent to positive union ($\overline{\cap} \equiv \cup$) of two databases, and therefore, their general algorithms are the same:

1. Initialize NDB_3 to NDB_1.
2. For every string $y \in NDB_2$, append y to NDB_3,

where NDB_3 holds the result of the operation. NDB_2 is constructed differently for each of the operators in θ and is discussed in detail below:

Negative-Equality ($\overline{\sigma}_{[\Upsilon]=v}$) Select

To implement Negative-Equality, it is sufficient to create a negative database that matches every binary string x for which $x[\Upsilon] \neq v$.

1. Initialize NDB_2 to the empty set.

2. For each position i in Υ:

   (a) Create a string x with the position indicated by the i^{th} entry of Υ set to the complement of the i^{th} bit in v (for clarity we assume that v has $|\Upsilon|$ bits) and set the rest of the positions to $*$.

   (b) Append x to NDB_2.

Theorem 1 *Negative Equality Select* ($\overline{\sigma}_{[\Upsilon]=v}$)*: A binary string x is matched in $NDB_2 \iff x[\Upsilon] \neq v$.*

Proof 3.2.1

1. Let x be a string matched in NDB_2 and y be the string that matches it. y must have been generated during the i^{th} iteration of the algorithm (step 2(a)). By construction, y differs from v in the position indicated by the i^{th} entry of Υ, therefore, y will not match any string exhibiting v, and, given that $y \not\asymp x$, $x[\Upsilon] \neq v$.

2. Let x be a binary string such that $x[\Upsilon] \neq v$, then x must differ in at least one position from v. Step 2(a) of the algorithm constructs a string y that matches x.

Negative-Less-than ($\overline{\sigma}_{[\Upsilon]<v}$) and Negative-Less-than-Equal ($\overline{\sigma}_{[\Upsilon]\leq v}$) Select

The following algorithm implements the Negative-Less-than operation. It creates a negative database that matches every binary string x for which $x[\Upsilon] \geq v$ (in terms of their binary integer values).

1. Initialize NDB_2 to the empty set.

2. For each bit i in v that is set to 0:

   (a) Create a string x of length l with the corresponding position set to 1, all positions to the left of it (more significant positions) set to 1 where v is 1, and all other positions set to $*$.

   (b) Append x to NDB_2.

3. Create a string y of length l for which the positions indicated by Υ have the same value as the corresponding positions in v, and all remaining positions set to $*$.

4. Append y to NDB_2.

Theorem 2 *Negative-Less-than Select* ($\overline{\sigma}_{[\Upsilon]<v}$)*: A binary string x is matched in* $NDB_2 \iff x[\Upsilon] \geq v$.

Proof 3.2.2

1. Let x be a binary string matched in NDB_2, and let $y \; \mathcal{M} \; x$. If y was generated in Step 2(a) then, by construction, $y[\Upsilon]$ and v do not match, and the most significant position, in which the strings represented by $y[\Upsilon]$ and v differ, is set to 0 in v and 1 in $y[\Upsilon]$; hence, all strings matched by $y[\Upsilon]$ are greater than v and $x[\Upsilon] > v$. Conversely, if y was generated in step 3 then $y[\Upsilon] = v$ and $x[\Upsilon] = v$. Therefore, $x[\Upsilon] \geq v$.

2. Let x be a binary string such that $x[\Upsilon] \geq v$.

   Assume $x[\Upsilon] > v$ and let i be the first position of x, from left to right, for which $x[\Upsilon]$ has a 1 and v has a 0. Step 2(a) creates a string y, such that $y \; \mathcal{M} \; x[\Upsilon]$ by setting every position to the left of i to 1 where v and $x[\Upsilon]$ have a 1; position i to 1 where v has as 0 and $x[\Upsilon]$ a 1, and all remaining positions to $*$.

Assume that $x[\Upsilon] = v$, Step 3 of the algorithm generates a string y for which $y[\Upsilon] = v$ and the rest of the positions are set to $*$, thus matching x. Therefore, there is a string in NDB_2 that matches x.

The algorithm to implement the Negative-Less-than-Equal operation can be derived from the algorithm presented above by removing step 3. Due to this similarity, its formulation and proof are omitted.

Negative-Greater-than ($\overline{\sigma}_{[\Upsilon]>v}$) and Negative-Greater-than-Equal ($\overline{\sigma}_{[\Upsilon]\geq v}$) Select

The algorithms for these operations are analogous to their $\overline{<}$ and $\overline{\leq}$ counterparts. The following algorithms with the necessary adjustments are presented, but their proofs are omitted due to their similarity to Proof 3.2.2. The following algorithm creates a database that matches every binary string x such that $x[\Upsilon] \leq v$ (in terms of their binary integer values).

1. Initialize NDB_2 to the empty set.
2. For each bit i in v set to 1:
   (a) Create a string x of length l with the corresponding position set to 0, all positions to the left of it set to 0 where v is 0, and all other positions set to $*$.
   (b) Append x to NDB_2.
3. Create a string y of length l for which the positions indicated by Υ have the same value as the corresponding positions in v, and the remaining positions are set to $*$.

4. Append y to NDB_2.

The algorithm to implement the Negative-Less-than-Equal operation can be derived from the algorithm presented above by deleting step 3.

If we assume that a string can be created in time proportional to its length l and that it takes c time to copy a string, then the Negative Select operation takes $O(cl^2 + |NDB|)$ time. The $|NDB|$ factor is due to the copying of the negative databases to create a separate output database. If the selection criteria, represented by NDB_2, is simply appended to NDB_1, the running time of the operation is reduced to $O(cl^2)$.

To summarize, restricting the contents of a positive database based on the selection criteria is accomplished by adding elements to its negative image. The result of the selection predicate can be viewed as a negative database itself, capable of being swapped in and out without interfering with how the database is queried.

For example, suppose $CustomerNDB$ represents a list of existing customers and their credit card number (CCN) (<name, CCN> tuples). Using the negative representation prevents sales clerk or order takers from browsing the database to obtain CCNs. As a marketing tool, if the purchaser is a member of $CustomerNDB$, she should be offered a promotion at checkout (the company has found that by offering promotions encourages repeat business). Management may also choose to offer additional promotions to current customers that hold a specific brand of credit card. This can easily be accomplished, using the operations described above, by restricting (using Select) $CustomerNDB$ to only those records that do exhibit a Visa credit card. This can be done by inserting records to $CustomerNDB$ that match every possible <name, CCN> pair for Visa credit card numbers. Note that the prefix of a credit card number, known as the bank identification number, identifies the credit card issuer network. In this example, a single entry with all positions set to $*$, except

the ones corresponding to the CCN's prefix (set to Visa's prefix, the number 4) will match all the desired strings. The way in which the promotional database is used does not change—if a <name, CCN> pair is found, then a promotion is offered. When the promotion for Visa card holders expires, the negative records are readily available and can be deleted.

3.2.2 Negative Union

We now turn to an operation that is trivially implemented using positive databases, but it requires more care when using negative databases. For the purpose of this section, we assume that the length of strings in both databases is the same. A new operation is defined over ternary strings called *coalesce* in order to correctly extract only the valid strings.

Definition 3.2.2 (Coalesce, $\odot$). Two strings x and y of length l coalesce into string z, $(z = x \odot y)$, if and only if $x \ \mathcal{M} \ y$ and $\forall i \ 1 \leq i \leq l$:

$$z[i] = \begin{cases} x[i], \text{ if } (x[i] = y[i]) \vee (y[i] = *) \\ y[i], \text{ if } x[i] = * \end{cases}$$

We now proceed to define negative union by first reviewing positive union, which can be expressed as:

$$DB_1 \cup DB_2 = \{x \mid x \in DB_1 \ \vee \ x \in DB_2\}$$

And, its complement is written as:

$$U \setminus (DB_1 \cup DB_2) = \{x \mid x \notin DB_1 \ \wedge \ x \notin DB_2\}$$

Then, the Negative Union, $\overline{\cup}$, can be defined as:

$$NDB_1 \overline{\cup} NDB_2 = \{z \mid z = x \odot y,\ x \in NDB_1,\ y \in NDB_2\}$$

The following algorithm produces a new negative database, NDB_3, that realizes $NDB_1 \overline{\cup} NDB_2$. See Table 3.3 for an example.

1. Initialize NDB_3 to the empty set.
2. For every $x \in NDB_1$:
   (a) For every $y \in NDB_2$, $x \ \mathcal{M} \ y$:
      i. Create a string $z = x \odot y$.
      ii. Append z to NDB_3.

Theorem 3 *Negative Union* ($\overline{\cup}$)*: A binary string* $x \in U \setminus (DB_1 \cap DB_2) \iff \exists x'((x' \in NDB_1 \overline{\cup} NDB_2) \ \wedge \ (x' \ \mathcal{M} \ x))$.

Proof 3.2.3

1. Let x be a string in $U \setminus (DB_1 \cup DB_2)$, then $x \in U \setminus DB_1$ and $x \in U \setminus DB_2$. By the definitions of NDB_1 and NDB_2, there is a string $x' \in NDB_1$, and a string $y \in NDB_2$ that match x; by transitivity, $x' \ \mathcal{M} \ y$. Step i of the algorithm creates a string z such that $z = x' \odot y$.

   Given that $z \ \mathcal{M} \ x'$ and $x' \ \mathcal{M} \ x$, it follows that $z \ \mathcal{M} \ x$. Therefore, there is a string in $NDB_1 \overline{\cup} NDB_2$ that matches x.

2. Let x be a binary string matched by some entry z in $NDB_1 \overline{\cup} NDB_2$. By construction, there are strings $x' \in NDB_1$ and $y \in NDB_2$, that match z and by transitivity, also match x. By the definitions of NDB_1 and NDB_2, $x \notin DB_1$ and $x \notin DB_2$. Therefore, $x \in (U \setminus (DB_1 \cup DB_2))$.

This operation is useful in situations where at least one of the negative databases is hard-to-reverse. Union provides a means to combine negative databases without having to reverse them beforehand—a privacy-enhancing feature.

3.2.3 Cartesian Product, Join and Intersection

This section discusses three closely related database operations. Cartesian product takes every string from the databases and combines them to produce all possible ordered concatenation of the strings. The complete relational algebra, as defined in [20], does not include join and intersection since they can be derived from other base operations. However, we include them because they are closely related to Cartesian product and their implementation, using negative databases, exhibits some interesting subtleties.

Let the generic symbol `op` denote one of the following three operators:

1. $\times$: Cartesian Product, no positions of interest defined, $\Upsilon_2 = \Upsilon_1 = \emptyset$
2. $\bowtie$: Join, when $|\Upsilon_1| = |\Upsilon_2|$, $0 < |\Upsilon_1| < |\Omega_m|$
3. $\cap$: Intersection, when $|\Upsilon_1| = |\Upsilon_2| = |\Omega_m|$

The universe over which each operation is defined is restricted by the positions defined by Υ_1 and Υ_2, and by the properties a string has with respect to these positions:

$$U_{l+m-|\Upsilon_2|} = \{xy \mid x[\Upsilon_1] = z[\Upsilon_2], y = z[\Omega_m - \Upsilon_2], x \in U_l, z \in U_m\} \tag{3.1}$$

The Cartesian product, Join and Intersection of two sets can be written as:

$$DB_1 \texttt{ op } DB_2 = \{xy \mid xy \in U_{l+m-|\Upsilon_2|}, y = z[\Omega_m - \Upsilon_2], (x \in DB_1 \wedge z \in DB_2)\}$$

The complement of these operations, referred to as the Negative Cartesian product ($\overline{\times}$), Negative Join ($\overline{\bowtie}$), and Negative Intersection ($\overline{\cap}$), can be defined as:

$$U_{l+m-|\Upsilon_2|} \setminus (DB_1 \mathtt{op} DB_2) = \{xy | xy \in U_{l+m-|\Upsilon_2|}, y = z[\Omega_m - \Upsilon_2], (x \notin DB_1 \vee z \notin DB_2)\}$$

The sets U_l, U_m, DB_1 and DB_2 are defined over the binary alphabet $\{0, 1\}$. A negative databases NDB_1, on the other hand, is defined over Σ—recall that a string x with $b_i = *$ represents two strings. In this way, a string with k $*$ symbols represents 2^k binary strings.

3.2.4 Negative Cartesian Product

The Negative Cartesian product is defined using regular expressions as:

$$NDB_1 \overline{\times} NDB_2 = \{x *^m \mid x \in NDB_1\} \cup \{*^l y \mid y \in NDB_2\}$$

This set can be constructed as follows:

1. Initialize NDB_3 to the empty set.
2. For every string $x \in NDB_1$, construct a string z that is prefixed by x and suffixed by m $*$ symbols. Append z to NDB_3.
3. For every string $y \in NDB_2$, construct a string z that is prefixed by l * symbols and suffixed by y. Append z to NDB_3.

Theorem 4 *Negative Cartesian Product ($\overline{\times}$): A binary string $x \in (U_{l+m} \setminus (DB_1 \times DB_2)) \iff \exists x'((x' \in NDB_1 \overline{\times} NDB_2) \wedge (x' \ \mathcal{M} \ x))$.*

Proof 3.2.4

Let xy be a binary string in $U_{l+m} \setminus (DB_1 \times DB_2)$, with $|x| = l$ and $|y| = m$.

1. Either $x \in (U_l \setminus DB_1)$ or $y \in (U_m \setminus DB_2)$. If $x \in (U_l \setminus DB_1)$ then, by the definition of NDB_1, there is a string $x' \in NDB_1$ than matches x; step 2 of the algorithm will create a string, $x'*^m$ matching xy. Conversely, if $y \in (U_m \setminus DB_2)$, there is a string $y' \in NDB_2$, $y'\ \mathcal{M}\ y$, and Step 3 will generate $*^l y$ matching xy. Therefore, xy is matched by some entry in $NDB_1 \overline{\times} NDB_2$.

Conversely, let xy be a binary string matched by some entry $x'y'$ in $NDB_1 \overline{\times} NDB_2$, with $|x| = |x'| = l$ and $|y| = |y'| = m$.

1. By construction, either $x' \in NDB_1$ or $y' \in NDB_2$. If $x' \in NDB_1$ then, by the definition of NDB_1, $x \notin DB_1$; likewise, if $y' \in NDB_2$ then $y \notin DB_2$. Therefore, $xy \in (U_{l+m} \setminus (DB_1 \times DB_2))$.

Examples are shown in Table 3.2 and Table 3.3. In summary, Negative Cartesian Product appends $*$ symbols to each record in both negative database, as suffix to NDB_1 and as prefix to NDB_2. A new negative database is created by combining the two sets.

3.2.5 Negative Join

The Negative Join of DB_1 and DB_2, using NDB_1 and NDB_2, is constructed by creating the appropriate mapping of the string positions specified in the join condition Υ_1 and Υ_2 (see Table 3.2 and 3.3 for an example).

$$NDB_1 \bar{\bowtie} NDB_2 = \{x *^{m-|\Upsilon_2|} | x \in NDB_1\} \cup \{wz | w[\Upsilon_1] = y[\Upsilon_2], w[\Omega_l - \Upsilon_1] = *^{|\Omega_l - \Upsilon_1|}, z = y[\Omega_m - \Upsilon_2], y \in NDB_2\}$$

1. Initialize NDB_3 to the empty set.
2. For each string $x \in NDB_1$, create a string z with x as its prefix and $(m - |\Upsilon_2|)$ $*$'s as its suffix. Append z to NDB_3.
3. For every string $y \in NDB_2$:
   (a) Create a string $w = *^l$ and map onto it the values of the join positions of y: for all i, set the value of w at the string position indicated as the i^{th} entry of Υ_1, to the value at the string position indicated in the i^{th} entry of Υ_2 of y.
   (b) Create a string z of length $m - |\Upsilon_2|$ by mapping onto it all the non-join positions of y in the following way: for all i, set the value of the i^{th} position of z to the value of the position indicated in the i^{th} entry of $\Omega_m - \Upsilon_2$.
   (c) Concatenate w with z and append to NDB_3.

The strings in the result of negative join will either have their prefix in $U \setminus DB_1$ suffixed with every possible $y = z[\Omega_m - \Upsilon_2], z \in U_m$ (see step 2). Or, it will have as suffix $y = z[\Omega_m - \Upsilon_2]$ (step 3(a)) prefixed with every possible string $x[\Upsilon_1] = z[\Upsilon_2], z \in (U \setminus DB_2)$ (step 3(b)).

Theorem 5 *Negative Join ($\overline{\bowtie}$): A binary string $x \in U_{l+m-|\Upsilon_2|} \setminus (DB_1 \bowtie DB_2) \iff \exists x'((x' \in NDB_1 \overline{\bowtie} NDB_2) \wedge (x' \ \mathcal{M} \ x))$.*

Proof 3.2.5

Let wz be a binary string in $U_{l+m-|\Upsilon_2|} \setminus (DB_1 \bowtie DB_2)$, with $|w| = l$ and $|z| = m - |\Upsilon_2|$. Then, either $w \notin DB_1$ or $z' \notin DB_2$, for $z = z'[\Omega_m - \Upsilon_2]$ and $w[\Upsilon_1] = z'[\Upsilon_2]$.

1. If $w \notin DB_1$, then w is matched by some $w' \in NDB_1$ and z' may be matched in NDB_2. Step 2 of the algorithm creates a string $w'*^{|\Omega_m - \Upsilon_2|}$ that will match wz. Therefore, wz is matched by some string in $NDB_1 \overline{\bowtie} NDB_2$.

2. If $z' \notin DB_2$, then z' is matched by some $z'' \in NDB_2$ and w may be matched in NDB_1. Step 3(b) creates string y, such that $y = z''[\Omega_m - \Upsilon_2]'$, that matches z, and step 3(a) creates a string x, that represents every binary string x' for which $x'[\Upsilon_1] = z''[\Upsilon_2]$ (a characteristic of all strings in $U_{l+m-|\Upsilon_2|}$, (see eq. 3.1)), that matches w. Step 3(c) creates string xy that matches wz. Therefore, wz is matched by some string in $NDB_1 \overline{\bowtie} NDB_2$.

Conversely, let wz be a binary string and xy an entry in $NDB_1 \overline{\bowtie} NDB_2$ that matches wz, where $|w| = |x| = l$ and $|z| = |y| = m - |\Upsilon_2|$.

1. If xy was generated by step 2, then $x \in NDB_1$ and, by the definition of NDB_1, $w \notin DB_1$. Substring $y = *^{m-|\Upsilon_2|}$ matches string $z''[\Omega_m - \Upsilon_2]$ for $z'' \in U_m$ where $z''[\Upsilon_2] = w[\Upsilon_1]$, and $z''[\Omega_m - \Upsilon_2] = z$. Therefore, $wz \in (U_{l+m-|\Upsilon_2|} \setminus (DB_1 \bowtie DB_2))$.

2. If xy was generated in step 3, then $w \,\mathcal{M}\, x$ and $w \,\mathcal{M}\, z'[\Upsilon_2]$ (step 3(a)). If $z' \in NDB_2$, then $z \,\mathcal{M}\, y$ and $z \,\mathcal{M}\, z'[\Omega_m - \Upsilon_2]$ (step 3(b)). Hence, there is a string $z'' \in U_m$, matched by z', such that $z''[\Upsilon_2] = w[\Upsilon_1]$ and $z''[\Omega_m - \Upsilon_2] = z$. By the definition of NDB_2, $z'' \notin DB_2$. Therefore, $wz \in U_{l+m-|\Upsilon_2|} \setminus (DB_1 \bowtie DB_2)$.

Notice that the Negative Join operation is similar to the Negative Cartesian Product, except that it shrinks the record length by keeping only one copy of the columns of interest specified by the join condition.

3.2.6 Negative Intersection

The Negative Intersection is defined as:

$$NDB_1 \overline{\cap} NDB_2 = \{x \mid x \in NDB_1\} \cup \{y \mid y \in NDB_2\},$$

which can be constructed using the following algorithm:

1. Initialize NDB_3 to NDB_1.
2. For every string $y \in NDB_2$, append y to NDB_3.

Theorem 6 *Negative Intersection* ($\overline{\cap}$)*: A binary string* $x \in (U_{l+m-|\Upsilon_2|} \setminus (DB_1 \cap DB_2)) \iff \exists x'((x' \in NDB_1 \overline{\cap} NDB_2) \wedge (x' \ \mathcal{M} \ x))$.

Proof 3.2.6

1. Let w be a string in $U_{l+m-|\Upsilon_2|} \setminus (DB_1 \cap DB_2)$, then $w \notin DB_1$ or $w \notin DB_2$. By the definitions of NDB_1 and NDB_2 there is a string $x \in NDB_1$ or a string $y \in NDB_2$ that matches w. The algorithm includes all strings in NDB_1 and NDB_2, thereby ensuring that there is a string in $NDB_1 \overline{\cap} NDB_2$ that matches w.
2. Let w be a binary string and x a string in $NDB_1 \overline{\cap} NDB_2$ that matches it. By construction, x either belongs to NDB_1, or to NDB_2: if $x \in NDB_1$ then, by the definition of NDB_1, $w \notin DB_1$; likewise, if $x \in NDB_2$ then $w \notin DB_2$. Therefore, $w \in (U_{l+m-|\Upsilon_2|} \setminus (DB_1 \cap DB_2))$.

An example is shown in Table 3.2 and 3.3. Negative Intersection is the simplest of all operators, because it applies de Morgan's law and computes the union of the two negative databases by combining the sets. Although not required, exact string duplicates may be eliminated, if desired.

3.2.7 Negative Project

The positive Project operator is a unary operation on a database that returns a subset of the attributes of a relation. The attributes can be viewed as the column values for the string positions specified in the order requested. If one views the Select operator as taking a horizontal slice of a relation, then Project takes a *vertical* slice. In the positive DB representation, Project outputs the value of all attributes within this vertical slice. However, in the negative database representation, Project is not simply the same vertical slice but its complement. It is the set of attributes not represented in the positive vertical slice. Therefore, a substring is in the result of a negative projection if and only if all possible strings of length l with the specified attributes missing from DB.

If we define positive Project as:

$$\pi_{\Upsilon}(DB) = \{x \in U_{|\Upsilon|} \mid \exists z \in U_{|\Omega-\Upsilon|}(\exists y \in DB(z = y[\Omega - \Upsilon], x = y[\Upsilon]))\}$$

Then, its complement is:

$$U_{|\Upsilon|} \setminus \pi_{\Upsilon}(DB) = \{x \in U_{|\Upsilon|} \mid \forall z \in U_{|\Omega-\Upsilon|}(\exists y \notin DB(z = y[\Omega - \Upsilon], x = y[\Upsilon]))\}$$

Using a negative database that represents the complement of DB, we write the Negative Project as:

$$\overline{\pi}_{\Upsilon}(NDB) = \{x \in U_{|\Upsilon|} \mid \ \forall z \in U_{|\Omega-\Upsilon|}(\exists y \in NDB(z \ \mathcal{M} \ y[\Omega - \Upsilon], x \ \mathcal{M} \ y[\Upsilon]))\}$$

Unlike the previous operations, there is no polynomial time algorithm that takes as input any NDB and outputs an NDB' that represents the Negative Project of NDB unless $\mathcal{P} = \mathcal{NP}$, see proof below. For a special case, a heuristic algorithm, Negative Reduce, has been implemented, see Section 3.4.

Theorem 7 *Negative Project ($\overline{\pi}$): A polynomial time algorithm for computing Negative Project implies $\mathcal{P}$=$\mathcal{NP}$.*

To show that negative project, $\overline{\pi}$, is $\mathcal{NP}$-Complete we first restate the definition of Non-Empty Self Recognition ($NESR$) shown to be $\mathcal{NP}$-Complete in [41]. Then, we use $NESR$ to show that there is no polynomial time algorithm for computing Negative Project unless $\mathcal{P} = \mathcal{NP}$.

Definition 3.2.3 Non-empty Self Recognition (NESR):
INPUT: A set NDB over Σ^l.
QUESTION: Is DB nonempty? In other words, is there some string in $U = \{0,1\}^l$ not matched by NDB?

Proof 3.2.7
Assume there is a polynomial time algorithm $\mathcal{M}$ that takes as input a negative database NDB and a bit position indicator Υ and it outputs $\overline{\pi}_{\Upsilon}(NDB)$.

We construct a polynomial time algorithm for NESR in the following way: given any instance of NESR with input NDB, call $\mathcal{M}$ with NDB and $\Upsilon = \{1\}$. If the resulting negative database matches strings s1=0 and s2=1 (one-bit long strings) answer "No", otherwise answer "Yes." We have just created a non-empty detector for NDB and answered NESR efficiently, using $\mathcal{M}$. Since NESR is $\mathcal{NP}$-Complete, then $\mathcal{P}$=$\mathcal{NP}$.

Intuitively, Negative Project is hard because a negative database contains all possible combinations of attribute values, except those that appear in DB. The only way an attribute value is not in NDB is if it appears with every other possible value of the *remaining* attributes in the positive database.

Even though Negative Project is not an efficient operation in general, it can be implemented for some special cases. The Negative Project of a negative database onto attribute Υ can be defined with respect to a fixed value v of the remaining attributes $\Omega - \Upsilon$. This is the negative version of selecting all records from DB that have v in $\Omega - \Upsilon$ and then projecting onto Υ. It can be accomplished by joining (using the positive equijoin) NDB with a table that has as its single entry v, and then removing all string positions except Υ (projecting onto Υ). The first part of the operation preserves all and only the NDB entries that match strings in $U \setminus DB$ that have v; the second part reduces the universe of discourse over which strings are defined to all strings with v. An example of this special case is presented in Section 3.4.

One way to avoid doing Negative Project operations is to design our database so that each sensitive attribute results in a single negative database table. So, a single field value is represented by a table containing negative records. Then, each negative table is associated with other fields using some table identifier. In this way, there is no need to perform a negative projection within a negative table. Therefore, projection now corresponds to projecting the entire negative table, see Chapter 4 for more details.

3.2.8 Negative Set Difference

The positive Set Difference operator is a binary operation on two databases, DB_1 and DB_2, having the same record length and same order of attributes. The result is the subset of records that are in the first database, DB_1, but not in the second, DB_2.

$$DB_1 \setminus DB_2 = \{x \mid (x \in DB_1) \wedge (x \notin DB_2)\}$$

The complement:

$$U \setminus (DB_1 \setminus DB_2) = \{x \mid x \notin DB_1\} \cup \{y \mid y \in DB_2\}$$

Then, Negative Set Difference can be written as:

$$NDB_1 \overline{\setminus} NDB_2 = \{x \mid x \in NDB_1\} \,\overline{\cup}\, \{y \mid y \notin NDB_2\}$$

Similar to Negative Project, there is no polynomial time algorithm that, given as input NDB_1 and NDB_2, outputs a negative database that represents the Negative Set Difference of NDB_1 and NDB_2.

Theorem 8 *Negative Set Difference ($\overline{\setminus}$): A polynomial time algorithm for computing Negative Set Difference implies $\mathcal{P}$=$\mathcal{NP}$.*

We will proceed by constructing a polynomial time algorithm for the NESR problem (see Definition 3.2.3).

Proof 3.2.8

Assume a polynomial time algorithm $\mathcal{M}$ that takes as input two negative databases NDB_1 and NDB_2 and it outputs $NDB_1 \overline{\setminus} NDB_2$.

We construct a polynomial time algorithm for NESR in the following way: given any instance of NESR with input NDB, let $\mathcal{M}$ compute $NDB' = \emptyset \;\overline{\setminus}\; NDB$. If $NDB' = \emptyset$ then answer "No" otherwise answer "Yes." Note that if NDB represents an empty DB, then NDB matches all strings in U and NDB' will necessarily be empty. On the other hand, if NDB fails to match at least one string in U, then NDB' will contain at least one entry, and thus, be non-empty. Thus, $\mathcal{M}$ can be used to answer $NESR$ efficiently. Since NESR is $\mathcal{NP}$-Complete, $\mathcal{P}$=$\mathcal{NP}$.

As proven above, creating a negative database that represents the Negative Set Difference of two negative databases cannot be realized efficiently in general. Figure 3.1 illustrates this problem using a Venn diagram. Given only NDB_1 and NDB_2, in order to resolve strings inside the positive intersection of DB_1 and DB_2, using only NDB_1 and NDB_2, would require exponentially resolving all strings in either DB_1 or DB_2.

However, we can determine efficiently the membership of a particular string with respect to the Negative Set Difference. For example, string x is in the Negative Set Difference if and only if there is a string in NDB_1 that matches it or if there is no string in $NDB_1 \overline{\cap} NDB_2$ that matches it, as defined below:

$$\begin{aligned} x \in (NDB_1 \overline{\setminus} NDB_2) \quad &\equiv \quad x \notin (DB_1 \setminus DB_2) \iff \\ &\quad \exists y \in NDB_1 (y \,\mathcal{M}\, x) \;\vee\; \forall z \in NDB_2 (z \not\mathcal{M}\, x) \end{aligned}$$

Conversely,

$$\begin{aligned} x \notin (NDB_1 \overline{\setminus} NDB_2) \quad &\equiv \quad x \in (DB_1 \setminus DB_2) \iff \\ &\quad \forall y \in NDB_1 (y \not\mathcal{M}\, x) \wedge \exists z \in NDB_2 (z \,\mathcal{M}\, x) \end{aligned}$$

Therefore, membership of a given string can be determined efficiently, but correctly producing a new negative set from the two negative database using their set difference is hard. For example, given NDB_1 = {01*, *00, 11*} and NDB_2 = {000,011,10*,11*}, then their Negative Set Difference, $NDB_3 = NDB_1 \overline{\setminus} NDB_2$ = {01*, *00, 11*, 001}. So, NDB_3 represents all the strings in the positive that is the result of $DB_1 \setminus DB_2$. In order to create this set, all 2^3 strings are verified for membership, and since string "001" is missing from both NDB_1 and NDB_2, "001" is

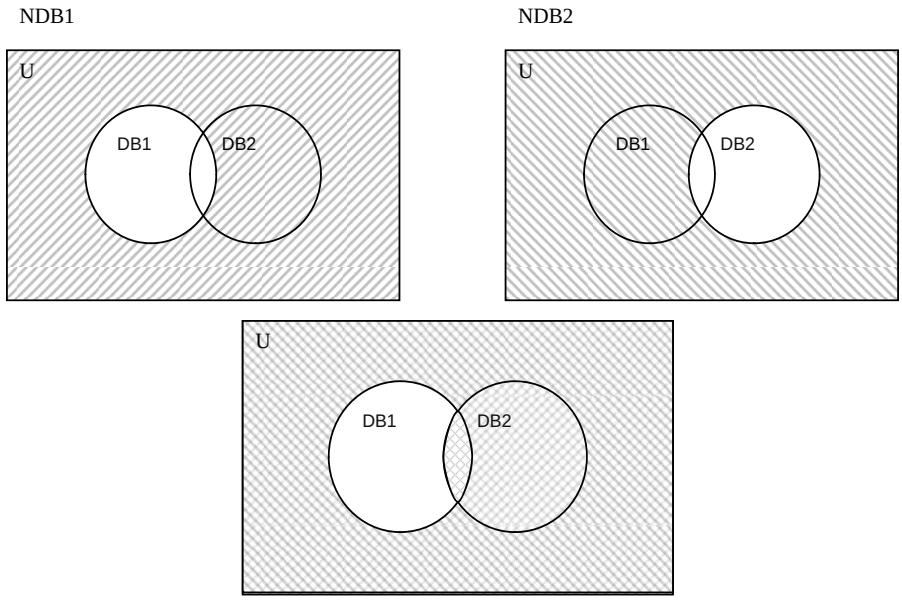

Figure 3.1: Given NDB_1 and NDB_2, $NDB_1 \bar{\setminus} NDB_2$ is shown by the shaded region.

included in the result. Notice that we can efficiently check if a specific string, say $x = 101$, is in their set difference without explicitly computing the entire NDB_3. First, verify that x is not represented by NDB_1. Since, x does not match any $y \in NDB_1$, we continue and check if there exists a string $z \in NDB_2$ such that $z \ \mathcal{M} \ x$. Finally, since $10^* \in NDB_2$, $10^* \ \mathcal{M} \ 101$, then $x \in (DB_1 \setminus DB_2)$.

The algorithms presented here were chosen for simplicity of exposition rather than for optimality; however, they suffice to illustrate the differences in complexities between a positive and a negative scheme. Table 3.4 gives the asymptotic time complexity for each operation under both positive and negative representation schemes. It assumes that a string can be created in time proportional to its length l. In addition, the time it takes to copy a string, c, is accounted for.

Operation	Positive DB	Negative DB
Select	$O(cl\|DB\|)$	$O(cl(l+\|NDB\|))$
$\bowtie$ and $\times$	$O(c(l+m)\|DB_1\|\|DB_2\|)$	$O(c(l+m)\|NDB_1\|+\|NDB_2\|)$
$\cap$	$O(cl\|DB_1\|\|DB_2\|)$	$O(cl(\|NDB_1\|+\|NDB_2\|))$
$\cup$	$O(cl(\|DB_1\|+\|DB_2\|))$	$O(cl\|NDB_1\|\|NDB_2\|)$

Table 3.4: Comparison of relational operators' asymptotic complexity. Negative Project and Negative Set Difference are $\mathcal{NP}$-Hard.

3.3 Example Scenario

Consider a law enforcement agency (LEA) investigating a money laundering scheme. It wishes to know which clients of data providers, Bank-1 and Bank-2, have carried out certain transactions for more than $10,000 and have also had relations with the currency exchange company (CEX), all during the month of June 2008. The banks and CEX are willing to provide information, but are concerned about the privacy of their clients; they are reluctant to hand over their entire client databases and would like to provide only the data needed for the investigation. Additionally, it is desired that the parties not communicate with one another or to remain ignorant of other participants in the same investigation.

Both providers can generate a table containing the client names and the transaction type for those individuals that have had operations for more than $10,000 during the month of June 2008: Bank-1 and Bank-2's tables contain tuples of the type <name, trans-1> and <name, trans-2> respectively. They each generate a hard-to-reverse negative database for their table: NDB_1 and NDB_2, and make that available to the LEA. For simplicity we assume that all the fields in all the databases follow some standard schema. The LEA wants to discover the names of the clients that withdrew more than $10,000 from Bank-1, deposited more than $10,000 in Bank-2, and also conducted business with CEX—CEX's table has tuples of the type <name>. The following SQL expression describes the desired operation:

```
SELECT Bank-1.name
    FROM Bank-1, Bank-2
        WHERE Bank-1.name=Bank-2.name and
            trans-1 = 'Withdrawal' and trans-2='Deposit'
    INTERSECT
        SELECT name
            FROM CEX
```

This query can be accomplished using the corresponding negative databases as follows:

1. Compute the Negative Join by <name> of NDB_1 and NDB_2, i.e., $NDB_1 \overline{\bowtie} NDB_2$.[1] This results in NDB_3 = <name, trans-1, trans-2>.

2. Generate a table, DB_{WD} of the tuples <trans-1, trans-2> with the single record {(Withdrawal, Deposit)}. This results in DB_{WD} = <trans-1, trans-2>.

3. Create the Natural Join of NDB_3 and DB_{WD} by <trans-1, trans-2>. This yields a negative database, NDB_{NJ}, of the tuples <name, trans-1, trans-2>. All entries have the trans-1 and trans-2 fields explicitly set to "Withdrawal" and "Deposit" respectively and are fully specified (no $*$ symbols appear at these positions).

4. Next, project NDB_{NJ} based on name by removing from NDB_{NJ} the fields trans-1 and trans-2. Notice that by joining NDB_3 and DB_{WD} we have fixed the transaction fields to specific values and thus effectively narrowed the universe

[1] Υ_1 and Υ_2 contain the string positions corresponding to the "name" field of both databases.

of discourse to names (character combinations) with those particular transactions (see Section 3.4 for details on this operation). Send the resulting negative database, NDB_P, to CEX. Note that the reverse of NDB_{NJ} will contain the names of clients that withdrew money from Bank-1 and deposited money in Bank-2—all information about other transactions has been eliminated. Therefore, the law enforcement agency can safely eliminate this two fields and send the resulting NDB to CEX.

5. Upon receipt, CEX computes the intersection of its client list and NDB_P by determining which of the names in its database (positive) is *not* in NDB_P. It returns the result to the LEA. Notice that CEX does not know what the resulting names refer to; the provenance of NDB_P's contents is unknown and the particular manipulations by the law enforcement agency—restricting transactions to deposits and withdrawals—have been erased.

The list of names received by the law enforcement agency represents the names of people that have withdrawn more than $10,000 from Bank-1, deposited more than $10,000 in Bank-2 and that have also transacted with CEX. The privacy of all other clients has been safeguarded and no direct communication was necessary between the entities being investigated.

It is worth mentioning that these databases are vulnerable to dictionary attacks because the space of possible names (and transactions) is relatively small. By using a longer identifier (other than name, credit card numbers), such an attack can be rendered intractable.

The operations illustrated in this section can be useful in other scenarios as well, without requiring the hard negative databases. For instance, the intersection of two positive databases can be accomplished by computing the Negative Intersection of their corresponding NDBs. The algorithm simply appends one negative database

to the other. If the NDBs are easy-to-reverse and the application requires that the original sets not be revealed, then the algorithm will be inadequate—the negative databases could be easily distributed (although the adversary must still guess where one ends and the other begins) and then consulted or reversed. In this case, the Negative Intersection could be created by randomly mixing the entries of both databases, the requirement being that no information links a particular item to a specific database. As described in the next section, the Morph operation, defined in [42], could also be used to mix the database records and de-identify strings.

3.4 Applications of Negative Relational Algebra

Test data shows that the space complexity of building large negative databases and applying the negative relational operators, is expensive. Strategies to ameliorate the problem includes distributed negative databases and using the Clean-Up operation introduced in [42] to eliminate redundant strings. Here, we describe some of the motivating applications for each strategy.

A prototype implementation of each of the relational operator algorithms specified in this chapter was developed by E.S. Ackley using Perl and C. It is available for download at `http://cs.unm.edu/~forrest/projects/ndb` under the GNU General Public License.

A motivating application for a distributed approach occurs in sensor networks. Next-generation sensor networks will likely involve active human participants that consume and divulge data to sensor networks. In such applications, privacy and confidentiality guarantees are desired. However, mechanisms and algorithms for privacy protection in sensor networks have been lacking. To address these concerns, Horey et al. developed and evaluated a set of protocols that enable anonymous data collection in a sensor network [59]. Sensor nodes, instead of transmitting their actual

data to a base station, transmit a data value that was not collected. The base station then uses these negative samples to reconstruct a histogram of the original data. These protocols are collectively referred to as a negative survey [39]. This approach could be extended so that each sensor contains a partial negative database, and the base station issues queries to retrieve information. For this to succeed, the relational operations presented in this chapter are essential for combining and manipulating the datasets.

A motivating application for the second strategy uses negative databases when the positive dataset approaches the power set of bit combinations. In this case, it can be more efficient to obtain an answer by working with the complement of the problem we intend to solve, and then complementing the solution. An example arises in logic program analysis [55]. Unlike the sensor network example, this class of problem is concerned less with privacy and more with producing the most compact representation. This insight opens up the possibility of using negative databases to extend the size of solvable problem instances and in answering questions that might otherwise be intractable. Helping solve this type of combinatorial problem is covered extensively in Chapter 5.

Using the program analysis example, we show how the Clean-Up algorithm (similar to the Compress algorithm described in Figure 5.3) helps control the size of the negative representation. Here, two relational operations, a modified Negative Equality Select and the Negative Union, are used to provide a practical negative projection. This restricted form of negative projection, known as Negative Reduce, iteratively decrements the size of NDB. Instead of projecting the bits-of-concern onto the negative database as if it were a positive representation, Negative Reduce selects the negative records for all the other bits, individually, once with the value of one, and again with a value of zero. These two partial negative databases, which no longer contain the original selected column, are then combined using Negative Union

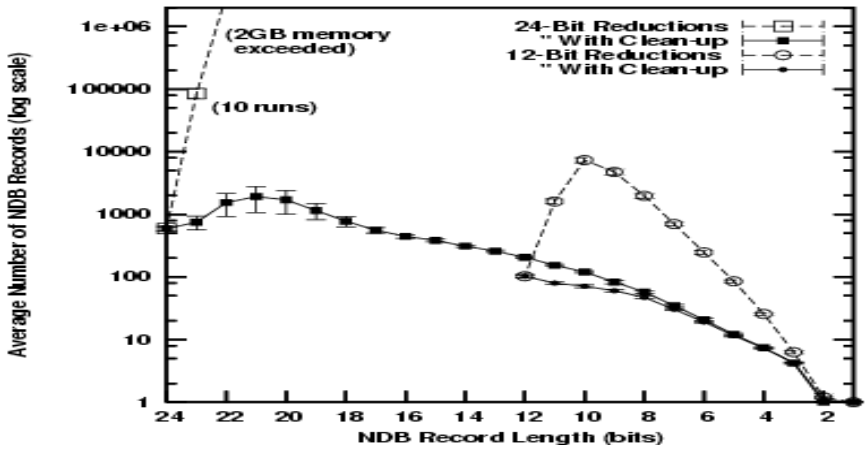

Figure 3.2: Average NDB size after single-bit Negative Reduce on 24- and 12-Bit length records, with and without Clean-Up.

followed by a constant number of Clean-Up and Morph operations. The Morph operation changes NDB records randomly without changing the equivalent positive database represented by introducing $*$'s randomly, if possible, while maintaining correctness [42]. The reduction is repeated until all of the unprojected bits have been processed. We compare the average resulting NDB size, with and without Clean-Up, at each iteration, eliminating the rightmost bit position until a single bit remains.

Based on 30 random databases each representing five 12-bit and 24-bit positive strings, the graph in Figure 3.2 shows nearly two orders of magnitude difference in the size of the negative database at the peak, occurring after the second bit reduction for the shorter length record. Their sizes converge as the number of solutions they represent diminishes. In the case of the longer record length, the simple (non-cleanup) approach fails due to memory constraints after the first or second bit reduction, while the projection with Clean-Up completes the test. Figure 3.3 shows the average NDB size, alternating between the Negative Reduce and Clean-Up operations, differs as much as three orders of magnitude.

The significant size difference is due to the use of "append" in the union step

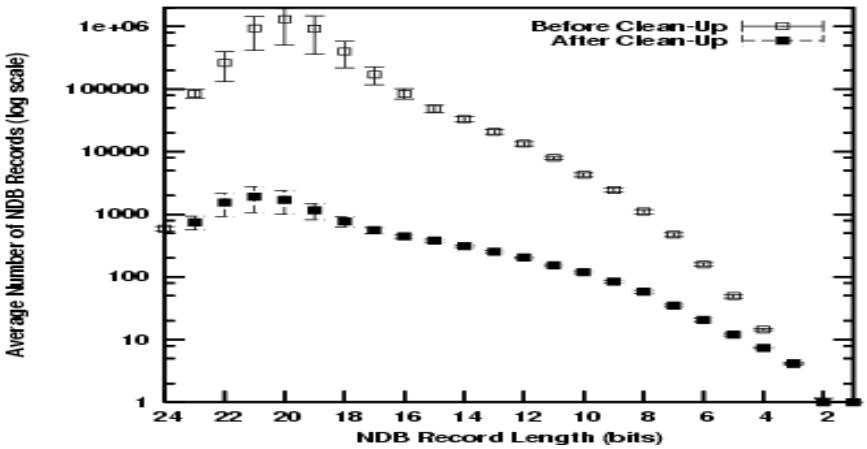

Figure 3.3: Average NDB size before and after Clean-Up in 24-Bit reduction test.

instead of using the Insert procedure, as discussed in [42] and shown in Figure 5.4. Insert modifies the entries being inserted into the NDB by adding $*$'s, if possible. Further tests show that Insert, while more cpu-intensive than simply appending records to the negative database, can maintain a more gentle growth in its size, within an order of magnitude of the size with Clean-Up on a 24-bit length record. These Clean-Up results closely fit the curves in Figures 3.2 and 3.3. If desired, further compression of NDB size can be achieved using ManagedGrowth shown in Figure 5.1.

Other applications may require the Select operation to be irreversible and/or for the restriction criteria itself to remain private. We can use the Compress operation shown in Figure 5.3 and the Insert procedure as means toward these objectives. Ensuring the privacy of the selection criteria can also be achieved by making the strings that comprise it into hard-to-reverse negative databases; some approaches to this, including a distributed architecture for singleton negative databases, are discussed in [45, 37] and Chapter 4.

By mitigating the impact of the relational operations on the NDB size with compression and distributed approaches, we are better able to apply the relational

algebra operations to manipulate the complemented data within its negative representation without specific knowledge of its contents enhancing the usability of negative databases.

3.5 Related Work

Negative databases and relational algebra are the two major sources from which this research draws. Negative databases are investigated in [44, 38], where it is shown that they can be created efficiently, its relation to SAT formulas is demonstrated, and its data hiding potential explained. The work in [37] investigates several applications and algorithms that share the same security assumptions and that are of immediate relevance to negative databases. Evidence regarding the construction of negative databases that can enhance privacy, in practice, in the absence of other cryptographic guarantees is provided in [43]. Reference [42] presents a series of operations that permit a negative database to be updated, i.e., that allows on-line changes to the contents of DB using only NDB. With regards to set and database theory, a relational algebra for positive databases is a well-developed area in practice [20]. Here, we rely only on the operations described in the original references, although further references on the relational algebra include [21, 30, 31, 32].

Other approaches to creating compact representations of Boolean functions and sets come from circuit minimization algorithms such as Karnaugh maps [66] and Quine-Mccluskey algorithm [95, 80] over truth tables. In addition, reduced ordered binary decision diagrams (ROBDDs) [13, 14] and its variants such as binary moment diagrams (BMD) [15] and zero suppressed binary decision diagram (ZBDD) [84] have been used to compactly represent binary functions. However, there exist functions for which the size is always exponential in the number of bits in the string. Among the differences between these approaches is the need of negative databases to always

obtain a compact representation of the complement of a set without explicitly calculating it, and the ease with which some operations can be performed. The negative relational algebra presented in this chapter will allow us to compare complexity of operations against these representations.

Other avenues for the application of NDB, stemming from the isomorphism of NDBs with logical formulas, include investigating SAT formulas themselves and strengthening the usefulness of SAT theory as it relates to other fields, such as constraint programming [62, 110, 10]. As discussed in Section 2.4, negative databases are also quite different than constraint databases.

Danezis et al. proposed an efficient "negative database" representation using cryptographic hash functions in [29]. Their technique generates a new random seed used to hash each positive data string to produce an obfuscated data pair, <random seed, hashed value>. They show that certain database operations and queries are possible using their scheme. We do not dispute any proven security properties of cryptographic hash functions. However, their process destroys the original strings losing the semantics of the original database. Their aim is to be able to answer membership queries given a specific string. It is also unclear how the relational operators will be performed to produce another negative database without this candidate string to restrict the database. For example, given a candidate string, their scheme can answer whether or not it is in the intersection of two database. However, it is unclear how the intersection of two hashed databases will be realized without maintaining the semantics of the original data in some way.

3.6 Summary and Conclusions

In this chapter, a stronger theoretical foundation for negative databases is presented bringing them closer to conventional relational databases. We increased our under-

standing of the complexities of working with the negative representation to assist in determining the spectrum of potential applications. The work here enabled the research leading to work on Set-sharing analysis using negative representations. Earlier research proposed a negative database framework supporting only membership queries and basic operations, such as insert and delete, to modify negative data. This chapter extends this earlier work by defining a complete and minimal relational algebra over the negative representation. The perspective proposed is that for each relational operator, the corresponding negative operator is defined such that the result of the negative operator applied to a negative representation will be equivalent to the positive version as if the operation is applied to the positive representation.

Specifically, a complete and closed set of relational operators for negative databases, select, union, Cartesian product (along with join and intersection), project, and set difference were described. We proved that no general efficient algorithm exists for the latter two operations but that efficient implementations are feasible for special interesting cases.

In addition, defining a negative relational algebra acts as a bridge to our understanding of negative databases' applicability in applications that may require mixing of both the positive and negative representations. We should design our database schema to avoid performing Negative Project and Negative Set Difference operations as much as possible. Furthermore, we should leverage the differences in complexity of the negative and positive relational operations to our benefit. Chapter 4 covers this issue in more detail.

Negative databases have been proposed as primitives for privacy-enhancing applications since some negative database constructions naturally limit the type of inferences that can be drawn from a dataset. The operations discussed here increase the versatility of negative databases by allowing the protected dataset to be manipulated in meaningful ways without diminishing its security. An agent can combine

two negative databases, restrict the contents of its positive image, and project onto a specific field without any knowledge of positive entries represented.

Further, the use of negative databases for non-secure applications is strengthened by having a relational algebra. In particular, we explore an application that needs to dynamically identify items that are not in its positive database and occasionally modify its contents. An operation such as Negative Select does not require access to the negative database other than for appending entries. Negative Select establishes conditions that the positive data must meet but requires no knowledge of the actual data, separating the ability to select a subset of the data from the need to own it.

There are several interesting avenues for future work. They include the optimization of the current relational algebra algorithms and their software implementation. Furthermore, extending the suite of operations beyond those presented will further our understand of the usefulness of negative databases in different applications. Comparative studies with other representations described above will prove useful and may expose applications where negative databases can also be used.

Chapter 4

Negative Database Management System

4.1 Introduction

This chapter proposes a negative database management system ($NDBMS$) that can effectively manage negative representation of information. To further advance negative database concepts and promote their use, I develop a straightforward and extensible architecture. The design accommodates both positive and negative representations simultaneously, in a single database management system, as a tool for private data sharing.

Current technology enables unprecedented ability for data mining and information retrieval among databases. With the global availability of data, many people are concerned with the possibility of information misuse and abuse by both legitimate businesses and illegitimate entities. Many news articles report several major breaches in security exposing millions of customers' private information to unauthorized personnel. Additionally, the rise of outsourcing non-core business functions,

i.e., database management services, has led to concerns about trust in the service provider's ability to protect and maintain confidentiality of their data. Furthermore, as a response to heightened privacy concerns and identity theft, laws mandate companies to ensure data privacy and to notify customers (and the public) of incidents that potentially exposes their private data [71, 72, 73, 74, 75, 76]. Such exposure leads companies vulnerable to loss of revenue from lawsuits, civil fines, and consumer confidence. These changes to the business environment are forcing the industry to reevaluate its approach to database privacy and security. However, the business requirement to share their data remains. Privacy preserving techniques, such as private matching, are mandatory precautions for prudent information sharing applications. These applications enable sharing of databases, possibly owned by different groups, without compromising sensitive information.

To take advantage of existing database systems, we design the $NDBMS$ over a relational database management system ($RDBMS$). Unlike previous work [38], the entire data record need not be in the negative format. Users select which attributes (fields within a record) are sensitive and convert them into their negative representations. Therefore, non-sensitive data may be left in the original positive form.

This chapter begins by motivating how negative databases can be used in data privacy scenarios. Then, we discuss related works that address the requirements posed by the scenarios. From these requirements, we present the $NDBMS$ architecture to help fulfill the requirements presented. Lastly, we analyze the design and conclude with a summary.

4.2 Application Scenarios

With the increased public dependence on electronic commerce and health care coupled with the costly spread of electronic-based fraud and identity theft [101], there is a strong demand for keeping databases private. Most privacy preserving protocols are based on cryptography [3, 50, 57, 53, 69]. Their solutions employ homomorphic encryption primitives to perform ciphered data matching. Others have proposed ways to conduct ciphered text searches for matching over databases [3, 8, 18, 52, 106]. However, performing other relational queries, i.e., joins, over encrypted data is cumbersome at best and not possible in many cases. We show that negative databases can be used for private data sharing.

In private sharing, the requester queries databases outside of its control. For example, suppose a group of retail companies, together with competitors, want to share their customer databases. The different companies agree to contribute data for a price or for similar database sharing privileges. They may want to obtain information to leverage its marketing strategies by offering special promotional sales with several companies. Now suppose, a specific retailer wants to find out which of its frequent-buyers shop at its competitors, using the same credit card. Using its own frequent-buyer list, the requester can pose queries against the intersection of all the credit card negative databases. A matching credit card number among its list of prime customers would provide this information. Other customers' information, those not in the requester's database, are not revealed. An advantage this scheme over cryptographic solutions is that, there are no keys to manage and new subscribers may join without having keys generated or registered.

If the requester wants to keep his query private and conduct private matching, then the sources can allow him to download the negative database and perform the query at the requester's location. A disadvantage of this approach is that the negative

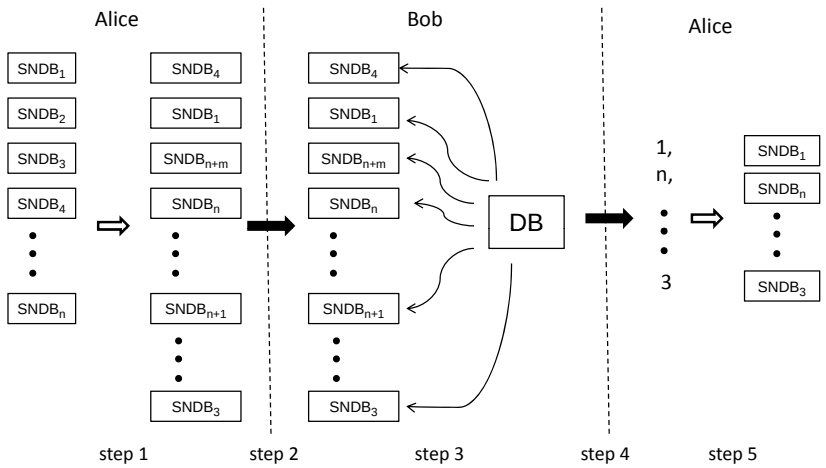

Figure 4.1: Private matching using negative databases.

database can get large in size, see Figure 4.5, which increases data transmission latency and query servicing.

Another way, initially proposed by Esponda, is for the requester to send all relevant negative databases to each data source for private matching. Figure 4.1 illustrates how to implement this private matching protocol. For example, Alice wants to find out which of her customers are in common with Bob. First, she extracts key attributes that can be used to identify a data record, such as social security number (SSN) or credit card number (CCN). Then, for each of these sensitive attributes, Alice converts then into a hard-to-reverse, singleton negative databases ($SNDB$) using the algorithm from [45]. Now, suppose Alice has n such attributes, private matching can be implemented as follows:

1. Alice creates m empty $SNDB$s ($DB = \emptyset$) with the same length as the original $SNDB$s. Alice assigns all $SNDB$s a unique, but temporary, identifier.

2. Alice sends all $n + m$ $SNDB$s (with identifiers) to Bob in random order.

3. Bob queries each $SNDB$ for a match using positive data values from his DB.

4. Bob returns the identifiers of negative databases that resulted in a positive match.

5. Alice receives the result and correlates which $SNDB$s, using the identifiers, were matched by Bob.

In this scenario, Alice's data is kept private in a hard-to-reverse $SNDB$ format. The potential for Bob browsing and spoofing the result is minimized by Alice introducing m empty $SNDB$s and randomizing the order of the $SNDB$s. As proven by Esponda [38], it is $\mathcal{NP}$-Complete to determine whether an NDB represents an empty DB or not. So, without reversing the $SNDB$s, Bob cannot tell which $SNDB$ is empty and which ones are legitimate. The $NDBMS$ can support this implementation and manage the large numbers of $SNDB$s effectively. Once again a large overhead is paid by having Alice transmit numerous $SNDB$s to Bob. However, the number of $SNDB$s to be transmitted is potentially much less than our first protocol above (where each source transmits its negative databases to the requester). To further improve efficiency, a compact representation for $SNDB$s becomes important.

4.3 NDBMS Architecture

In this section, we define the negative database management system architecture. For simplicity, we design the $NDBMS$ on top of a prominent relational database management system called PostgreSQL [94]. In this way, we can take advantage of built-in $DBMS$ management and optimization features, such as caching and query pre-planning. In addition, indexing and sorting of fields may be used to improve query performance. The following sections detail each area of the architecture shown in Figure 4.2.

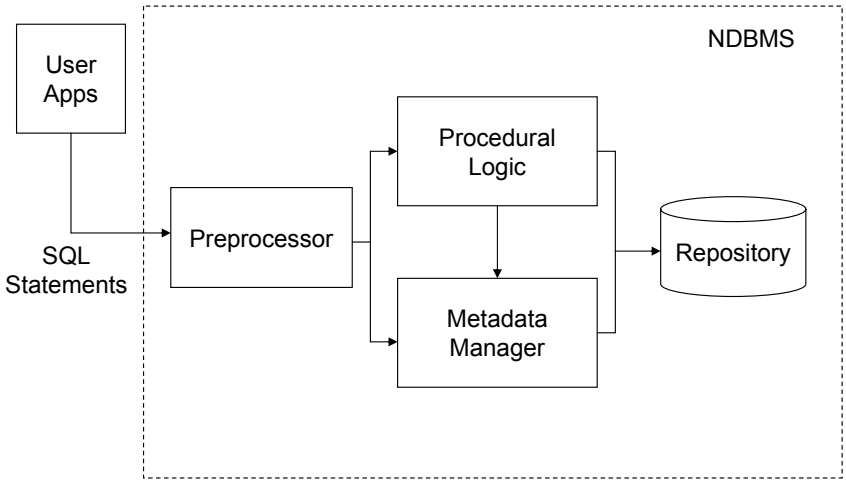

Figure 4.2: An architecture for a Negative Database Management System (NDBMS).

tablename	
b_1	char
b_2	char
.	
.	
.	
b_l	char

Table 4.1: Table scheme for Negative Database

4.3.1 Internal Data Representation

An important first step is to determine an appropriate internal negative data representation in a relational database paradigm. This internal representation will determine all other functional designs and implementations. For simplicity, I implemented a straightforward internal representation, i.e., where each bit value is stored as a character field in a table, as shown in Table 4.1. This representation can be queried for a *match* using the SQL query string below. The candidate string we want to match is kept in a table called QueryDB and the negative database is in NDB1.

```
SELECT true FROM QueryDB qdb
   WHERE EXISTS (SELECT true FROM NDB1 ndb1
      WHERE (qdb.b1 = ndb1.b1 OR qdb.b1 = '*' OR ndb1.b1 = '*')
        AND (qdb.b2 = ndb1.b2 OR qdb.b2 = '*' OR ndb1.b2 = '*')
        AND ...
        AND (qdb.bl = ndb1.bl OR qdb.bl = '*' OR ndb1.bl = '*'));
```

Since the negative representation for a typical positive database can be large, the efficiency of the negative data representation should be considered. An alternative internal representation, shown in Appendix A, maintains the location of each *specified* bit. This internal representation saves storage space since unspecified bits information are not stored. Although showing promise, further research showed that queries incur higher overhead in evaluating the unspecified bit values. More specifically, since all unspecified bits are resolved dynamically, the queries takes much longer. This slows down query executions to unacceptable levels (several minutes) even for records less than 100 bits long. In addition, using this representation, negative databases with large variance in the number of specified bits cannot be handled efficiently. As a result, in most cases, unspecified locations must be maintained anyway. Therefore, we resorted to the keeping each ternary bit value as a column value. Other representations should be considered in future research efforts. In any representation, one must evaluate the tradeoffs of a more compact representation with query execution time.

4.3.2 Database Schema

Previous work, using negative database, combine multiple fields as a single negative record. Although certain bit positions are designated as different attributes, the entire record is converted into the same negative format. If a single attribute required

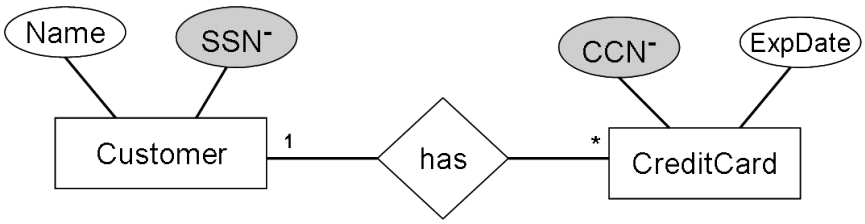

Figure 4.3: Example Entity-Relationship diagram.

strong protection, then the entire record must be transformed and maintained as negative database. However, most applications do not require the entire record to be protected. This only causes inefficient queries accessing specific attributes within a record.

Figure 4.3 illustrates an example entity-relationship diagram for a customer database containing the customer's name and SSN. This customer entity *has* zero or more credit cards. Each credit card entity has a number (CCN) and expiration date, registered with the company. The sensitive values are a customer's SSN and associated $CCNs$.

In this example, the CCN^- or SSN^- attributes, are converted to a hard-to-reverse $SNDB$. The non-sensitive data, Name and ExpDate, can be maintained in its original positive format. Alternatively, we can separate the non-sensitive attributes and convert them to an easy-to-reverse negative database using the Prefix algorithm [38] or NegConvert from Figure 5.4. Then, we would be able to reverse them easily. I selected to combine both positive and negative representation using a single system.

The relationships among records and fields can be maintained by assigning positive identifiers, such as PostgreSQL object identifiers ($OIDs$), to each entity. These identifiers are system generated assigned to each table in the database. Therefore, negative attributes, i.e., CCN^- shown in Figure 4.3, can be associated with the CreditCard entity using CreditCard's primary identifier (not shown), along with the

corresponding ExpDate.

This schema can be used to satisfy the scenario described earlier. Suppose Alice has a list of clients and their credit cards. She wants to know if Bob also has some of her clients without revealing any of her clients information. She sends the CCN^-'s in $SNDB$ format to Bob along with some empty $SNDB$s. Bob queries each one using a positive list of $CCNs$ and returns the identifiers of $SNDB$s that produced a match. From this result, Alice can tell which of her clients are also on Bob's list.

4.3.3 Design and Implementation

The current $NDBMS$ implementation provides a framework to conduct tests and analyze negative database operations, especially with respect to privacy preserving applications. Average users, familiar with relational databases and SQL commands, are able use negative databases with minimal learning.

Interface and Preprocessor

Users interact with the $NDBMS$ through a user application such as a graphical or web user interface. Queries are submitted through the interface in Structured Query Language (SQL) format. Upon receipt of the SQL statement, the Preprocessor parses the SQL statement and determines how to service the statement. The parsing process converts the ASCII values, into its binary equivalent and stores it in a temporary query table in the same internal negative format. From here, the SQL query may take one of two paths or both paths, through the Procedural Logic block and/or the Metadata Manager block. If the query can be serviced using metadata alone, i.e., database size, then the query is serviced solely by the Metadata Manager.

The current user interface is the existing user interface provided by PostgreSQL.

A graphical user interface called pgAdmin or a terminal interface are provided to conduct SQL queries. Query results are displayed on a terminal screen, saved onto a new table or file. Future implementation will provide a dedicated, platform independent user interface, such as a web or Java based front end. The interface should allow users to select whether or not the current attribute should be in the positive or negative format.

Currently, the Preprocessor uses several steps in order to save data into the negative format. A new value, say a credit card number, is converted into its binary representation and a singleton NDB is generated using existing code implemented by E.S. Ackley. It is available for download at `http://cs.unm.edu/~forrest/projects/ndb`. Then, a preprocessor implemented in C, parses a file in negative database binary format for inputting into a relational table format previously created. Future implementation will convert the positive text value and convert it into a negative database format for input into a new negative table automatically. Once the data is in a negative table format, queries can be levied against it.

Procedural Logic

One path a query may take is through the Procedural Logic block. It executes the necessary functions to service the incoming SQL query. Any change to contents of the database must go through this path. In addition, any queries that cannot be answered solely by the metadata goes through this path. The Procedural Logic also can make changes to the metadata by communicating with the Metadata Manager. Any change to the actual data contained or the structure of the database induces changes to the metadata. Preliminary tests showed that it is possible to implement basic NDB operations (Negative Pattern Generate, Insert and Delete) using an $RDBMS$. The relational operations described in Chapter 3, except for Negative Project and Negative Set Difference, were implemented using PostgreSQL's $PG/PLSQL$. In

addition, procedures for conducting membership queries are implemented. Future work should evaluate using a separate procedural language that can interface with PostgreSQL.

Metadata Manager

The other path a query may take is to go directly to the Metadata Manager. Certain basic queries that can be answered by the metadata need not consult the rest of the system. These queries are interested only about the metadata information, such as number or names of NDBs, size of an NDB, etc. These metadata information are maintained by the existing $RDBMS$. If additional metadata is required, then new metadata tables can be created and stored in a privileged area of the repository.

Currently, metadata maintained are those generated by PostgreSQL. No additional metadata specific to negative database format, i.e., number of unspecified bits, are maintained.

Repository

Positive and negative data are stored in the repository. It is a repository that provides automated record level identification using auto-incremented fields. It also provides negative data identification using object identifiers at table level. The repository is used to maintain consistency and help eliminate redundancy within the $NDBMS$. Negative databases can be very large, the chosen $RDBMS$ must be able to perform incremental loading of data from extremely large tables.

Figure 4.4 shows a steady linear growth in the length of time that membership queries are executed. The average query execution time are similar for data that matched and not matched in a negative database. Matching records takes more time

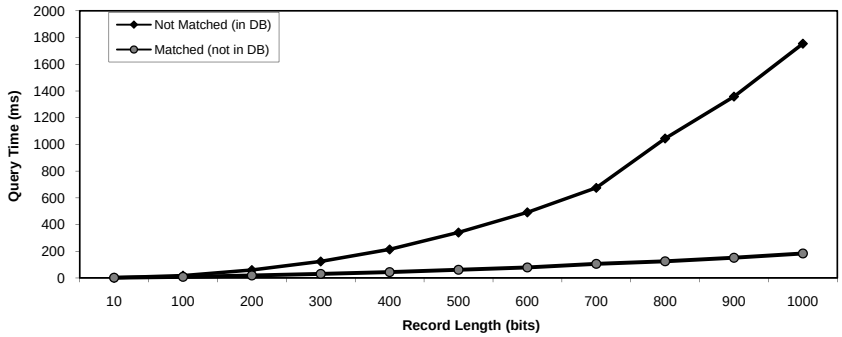

Figure 4.4: Average query times for finding a match between a single record and singleton negative database (of different lengths).

since it could not immediately reject the query, i.e., the rest of the bits (fields) are verified.

4.4 NDBMS Analysis

Current negative database implementations rely on individual files as a means of persistent storage. There is no systematic ability to maintain a database schema. Thus, attribute labeling and entity relationships must be done explicitly by each user. An $NDBMS$ can provide this capability automatically. A centralized data management can take advantage of availability of all data from a single perspective. A well designed schema will reduce data redundancies and inconsistencies [33]. Storing data in a central $DBMS$ also allows designers to better establish and enforce standards for data format, naming conventions, and database schema. Another advantage of a $DBMS$ is it promotes data independence from applications that use the data. Multiple applications can access the same data as required.

Simple applications do not necessarily need to use a $DBMS$. It will increase the overall database size, as shown in Figure 4.5, and may add unnecessary complexity

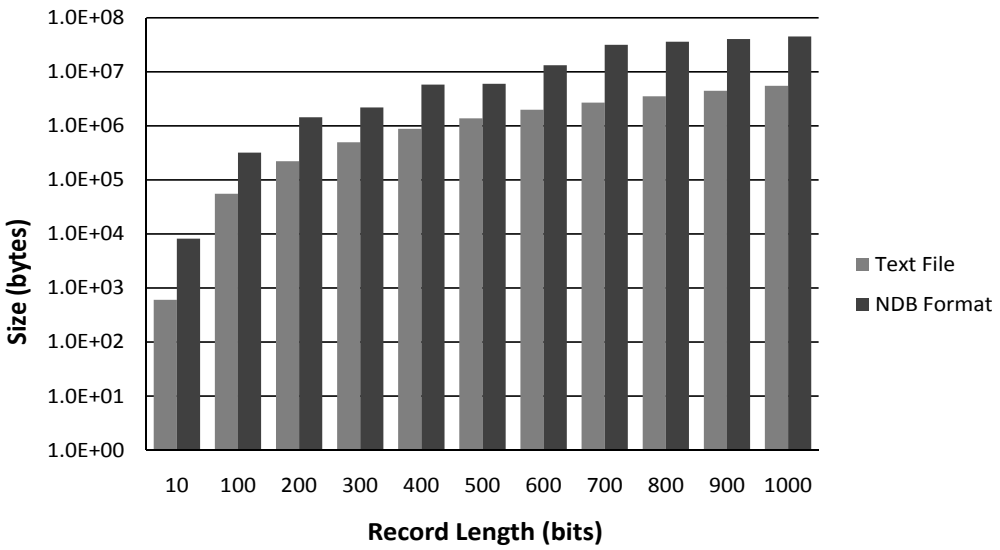

Figure 4.5: Comparison of negative database storage size as a text file and as a table in NDBMS.

to programs. Due to overhead associated with the $RDBMS$, i.e., metadata and internal bookkeeping data, the resulting negative tables are consistently larger in size than its text file equivalent.

Davida et al., studied several cryptographic techniques used to encrypt databases [34]. He described characteristics that database encryption systems should possess to ensure a practical database management system. We compare our $NDBMS$ to their guidelines.

• *Encryption system does not need to be theoretically secure...systems based on classically hard problems are sufficient.* A negative database, due to its correspondence to a SAT formula, satisfy this criteria directly. Although not cryptographically secure, reversing NDBs is $\mathcal{NP}$-Hard, in general. Therefore, it is important to be able to generate consistently hard-to-reverse NDBs. An algorithm to create hard-to-reverse $SNDB$s was presented in [45].

• *Encryption and decryption must not degrade performance to unacceptable levels, especially decryption, which is done more often during query execution.* Most privacy enhancing operations deal with comparing encrypted values, similar to membership queries. When using NDBs, membership queries remain efficient with respect to the input size, an example is shown in Figure 4.4.

• *Encrypted data must not have significantly greater storage volume than unencrypted data.* Negative databases will have less storage than its positive data only when the positive data is very large. Therefore, for single positive data, the size of the NDB created will be much larger, see Figure 4.5. Thus, hard-to-reverse, singleton NDBs violate this guideline.

• *Encryption must occur at the record level of a database. Since the position of a record is likely to change during its lifetime, there should be no need to decrypt $n-1$ records to get to the n^{th} record.* We propose to obfuscate data values at the attribute (not record) level. Our scheme facilitates other relational operations, i.e., Negative Project. Also, we are able to integrate both positive and negative representation within a single record. The positive data can help index the records to narrow down the search space. If an entire record and its individual attributes are obfuscated using NDBs, then each one will need to be queried individually. This will add to the query execution time.

• *An encrypted record must not contain a series of individually encrypted fields. Such a scheme can lead to pattern matching and substitution attacks.* As stated above, our scheme obfuscates at field (attribute) level, but maintains other fields in the positive. Each field is secured according to the hardness of individual $SNDB$s created. As an added protection, after a set amount of query access, we can trigger a Morph operation [38] to change the elements of a specific NDB, without changing the positive data represented.

• *Encryption should support logical view presentation depending on different user privileges.* Our proposed database schema enables a logical of view of the database using an entity-relationship diagram, such as the example shown in Figure 4.3. By building on top of a relational $DBMS$, the system similarly control access to each table based on user privileges imposed by the database administrator.

• *Reads and writes to the database must be allowed without any special constraints.* When using our $NDBMS$, reads and writes would occur as normal with the exception of calling customized procedures for the relational operations, rather than traditional SQL queries. By using an existing $RDBMS$, many of the underlying system operations, i.e., transaction management, remains unchanged.

• *The systems should be able to detect and reject data using false encryption key, even without knowing what the data contain.* Since our scheme does not use keys, this is not applicable to the proposed $NDBMS$ architecture.

As outlined above, integrating the negative and positive representation in a single system meets many of the criteria given by Davida et al. for using encryption in database systems. Our $NDBMS$ design meets many of their recommended guidelines making it practical for use by a wider community.

4.5 Conclusion

This chapter describes a database management system for negative representation of information. Having an efficient protocol that can be used for privacy preserving applications will encourage collaborations among entities that need to share data but who are unwilling to compromise data privacy. In addition, negative databases provide an alternative solution to a class of problems previously reserved for cryptography.

We described an extensible $NDBMS$ built upon an existing $RDBMS$. An $NDBMS$, general enough to accommodate both negative and positive relational data, will encourage the adoption of negative databases. Moreover, an easy-to-use and extensible infrastructure for working with negative data will encourage users to confidently share their private data. The benefit of an automated management system is clear from the user and administrator perspective. In addition, better research can be conducted using negative databases once negative operations are available and automatically managed by an $NDBMS$. Thus, it allows researchers to concentrate on higher level problems, such as efficient protocols for privacy preserving applications using negative databases. Without the availability of tools to help ease user interaction with negative databases, their use will be relegated to academia or a few truly interested parties.

This initial $NDBMS$ implementation can perform the relational algebra operations efficiently (other than the Negative Project and Negative Set Difference). We implemented a preprocessor to import a NDB file into a relational table. The implementation was used to study the additional storage required for storing an $SNDB$ file in a relational table. In addition, average membership query execution times for large NDBs were studied and resulted in acceptable levels.

In the future, we would like to implement the procedural logic algorithms using a single programming language, such as C, Java, etc. Any future implementation should strive to service queries more efficiently and store NDBs compactly. New internal representations and associated functionalities should be compared to current implementation. In addition, future research should incorporate native $RDBMS$ optimization functionalities, i.e., indexing, to improve overall performance.

Chapter 5

Efficient Representations for Set-Sharing Analysis

5.1 Introduction

This chapter discusses how concepts from negative databases (NDB) can help perform efficient set-sharing analysis in abstract interpretation of logic programs [27]. Abstract interpretation approximates the semantics of a program by making generalizations and is commonly used during static program analysis. The goal of static analysis, e.g., set-sharing analysis, is to conservatively and efficiently estimate the dynamic behavior of a program without executing it. Results from the analysis can be used for debugging and compiler level optimizations. However, due to the large number of variable combinations that must be evaluated during set-sharing analysis, an intractable number of sharings may occur. Due to memory constraints and/or long computation time during analysis, compromises are commonly made between precision of the analysis and its tractability. Therefore, any steps toward a more compact set-sharing representation and efficient, yet precise, analysis are desired.

This optimal solution increases the size of solvable set-sharing problem instances.

In close collaboration with J. Navas and E.S. Ackley, we studied two set-sharing representations. Navas initially posed the possibility of using negative database concepts to represent set-sharings in complement form. With the help of Ackley and her code implementations, I developed the majority of the algorithms and proofs. We showed that it was not only possible to use the negative representation but is more efficient than its positive binary string representation. Our collaborative efforts were the key to success to this research. Note that a large part of this chapter is excerpted from a paper accepted at the International Conference on Logic Programming 2008 [108].

Our primary goal was to develop a more efficient representation to accommodate larger problem instances. We show that our representations, ternary and negative ternary, perform precise set-sharing analysis using abstract unification more efficiently than strictly binary set-sharing representation. In the future, our work may be used to solve efficiently other combinatorial problems amenable to our proposed representations.

5.2 Preliminaries

In abstract interpretation of logic programs, sharing analysis has received considerable attention. Two or more variables in a logic program are said to *share* if in some execution of the program they are bound to terms that contain a common variable. A variable in a logic program is said to be *ground* if it is bound to a term that does not contain free variables. *Set-Sharing* is an important type of combined sharing and groundness analysis. It was originally introduced by Jacobs and Langen [61, 70] and its abstract values are sets of sets of variables that keep track, in a compact way, sharing patterns among variables.

Example 5.2.1 Set-Sharing encoded by set of sets of variables. Let $V = \{X_1, X_2, X_3, X_4\}$ be a set of variables of interest. The abstraction in *set-sharing* of a substitution such as $\theta = \{X_1 \mapsto f(U_1, U_2, V_1, V_2, W_1),\ X_2 \mapsto g(V_1, V_2, W_1), X_3 \mapsto g(W_1, W_1), X_4 \mapsto a\}$ will be $\{\{X_1\}, \{X_1, X_2\}, \{X_1,\ X_2, X_3\}\}$. Sharing group $\{X_1\}$ in the abstraction represents the occurrence of run-time variables U_1 and U_2 in the concrete substitution, $\{X_1, X_2\}$ represents V_1 and V_2, and $\{X_1, X_2, X_3\}$ represents W_1. Note that X_4 does not appear in the sharing groups because X_4 is *ground*. Note also that the number of (occurrences of) run-time variables shared is abstracted away.

Sharing has been used to infer several interesting properties and perform optimization and verification of programs at compile-time, most notably but not limited to: occurs-check reduction (e.g., [104]), automatic parallelization (e.g., [91, 90, 17]), and finite-tree analysis (e.g., [5]). The accuracy of Set-Sharing has been improved by extending it with other kinds of information, the most relevant being *freeness* and *linearity* information [89, 61, 90, 24, 58], and also information about *term structure* [90, 68, 12, 88]. Sharing in combination with other abstract domains has also been studied [23, 47, 25]. The significance of Set-Sharing is that it keeps track of sharing among sets of variables more accurately than other abstract domains such as *Pair-Sharing* [104] due to better groundness propagation and other factors that are relevant in some of its applications [16]. In addition, Set-Sharing has attracted much attention [22, 25, 6, 16] because its algebraic properties allow elegant encodings into other efficient implementations (e.g., *Reduced Ordered Binary Decision Diagrams*, *ROBDDs* [14]). In [91, 90], the first comparatively efficient algorithms were presented for performing the basic operations needed for implementing set sharing-based analyses.

However, Set-Sharing has a key computational disadvantage: the *abstract unification* (*amgu*, for short) implies potentially exponential growth in the number of sharing groups due to the *up-closure* (also called *star-union*) operation which is the

heart of that operation. Considerable attention has been given in the literature to reducing the impact of the complexity of this operation. In [113], Zaffanella et al. extended the Set-Sharing domain for inferring pair-sharing from a set of sets of variables to a pair of sets of sets of variables in order to support widening. The key concept is that the set of sets in the first component (called *clique*) is reinterpreted as representing all sharing groups that are contained within it. Although significant efficiency gains are achieved, this approach loses precision with respect to the original Set-Sharing. A similar approach is followed in [92] but for inferring set-sharing in a *top-down* framework. Other relevant work was presented in [78] in which the up-closure operation was delayed and full sharing information was recovered lazily. However, this interesting approach shares some of the disadvantages of Zaffanella's widening. Therefore, the authors refined the idea in [77] reformulating the amgu in terms of the *closure under union* operation, collapsing those closures to reduce the total number of closures and applying them to smaller descriptions without loss of accuracy. In [25], the authors show that Jacobs and Langen's sharing domain is isomorphic to the dual negative of *Pos* [4], denoted by $\overline{coPos}$. This insight improved the understanding of sharing analysis, and led to an elegant expression of the combination with groundness dependency analysis based on the reduced product of Sharing and Pos. In addition, this work pointed out the possible implementation of $\overline{coPos}$ through ROBDDs leading to more efficient implementations of Set-Sharing analyses.

In this research, we present a different approach in order to mitigate the computational inefficiencies of the *set-sharing* domain. We propose two representations that compress efficiently the number of elements into fewer elements enabling more efficient abstract operations without any loss of accuracy. The first representation, tSH, compacts the sharing relationships by eliminating redundancies among them. The second, $tNSH$, leverages the complement (or negative) sharing relationships of the original sharing set. Intuitively, let $sh_\mathcal{V}$ be a sharing set over the set of variables

of interest $\mathcal{V}$, then $tNSH$ keeps track of $\wp(\mathcal{V}) \setminus sh_{\mathcal{V}}$. This new capability of $tNSH$ dramatically reduces the number of elements to represent as the cardinality of the original set grows toward $2^{|\mathcal{V}|}$. It is important to notice that our work is not based on [25]. Although they define the dual negated positive Boolean functions, $\overline{coPos}$ does not represent the entire complement of the positive set. Moreover, they do not use $\overline{coPos}$ as a means of compacting relationships but as a way of representing Sharing through Boolean functions. We also represent Sharing through Boolean functions, but that is where the similarity ends.

In the remainder of this chapter, Section 5.3 first describes Jacobs and Langen's *set-sharing* domain, bSH, adapted for handling binary strings. Section 5.4 presents tSH, a more compact representation of bSH using ternary strings. Section 5.5 introduces a novel representation, $tNSH$, the complement (or negative) of the original set-sharing. Finally, experimental evaluations of these representations are shown Section 5.6, and a conclusion is given Section 5.7.

5.3 Set-Sharing Encoded by Binary Strings

The following presentation is based on the notation used by [113, 25] for abstract unification operations which are rather intuitive. However, for compactness, we represent the set-sharing domain as a set of strings rather than a set of sets of variables.

Definition 5.3.1 (Binary sharing domain, bSH). Let alphabet $\Sigma = \{0, 1\}$, $\mathcal{V}$ be a fixed and finite set of variables of interest in arbitrary order, and Σ^l the finite set of all strings over Σ with length l, $0 \leq l \leq |\mathcal{V}|$. Let $bSH^l = \wp^0(\Sigma^l)$ be the *proper power set* (i.e., $\wp(\Sigma^l) \setminus \{\emptyset\}$) that contains all possible combinations over Σ with length l. Then, the *binary sharing domain* is defined as $bSH = \bigcup_{0 \leq l \leq |\mathcal{V}|} bSH^l$.

Note that the bSH domain includes all sets of binary strings of length not greater than $|\mathcal{V}|$. However, given a particular instance $bsh \in bSH$, the following property always holds: $\forall s_i, s_j \in bsh$, length(s_i) = length(s_j). We will denote binary sets with the same length l through the regular expression $\{0,1\}^l$.

5.3.1 Notation

Let $\mathcal{F}$ and $\mathcal{P}$ be sets of ranked (i.e., with a given arity) functors of interest, e.g., the function symbols and the predicate symbols of a program. $Term$ is used to denote the set of terms constructed from $\mathcal{V}$ and $\mathcal{F} \cup \mathcal{P}$. Although somehow unorthodox, this allows for simply writing $g \in Term$ whether g is a term or a predicate atom, since all operations apply equally well to both classes of syntactic objects. $\hat{t}$ is denoted by the binary representation of the set of variables of $t \in Term$ according to a particular order among variables. For two elements $r, t \in Term$, $\hat{rt}$ represents the binary representation of the set of variables of r and t in order. Since $\hat{t}$ always will be used through a bitwise operation with some string of length l, the length of $\hat{t}$ must be l. If not, $\hat{t}$ is adjusted with 0's in those positions associated with variables represented in the string but not in t.

5.3.2 Abstract Operations

The following definitions are adapted from the standard definitions for the sharing domain [61] to accommodate our binary string representation:

Definition 5.3.2 (Binary relevant sharing $rel(bsh, t)$ and irrelevant sharing $irrel(bsh, t)$). Given $t \in Term$, the set of binary strings in $bsh \in bSH^l$ of length l that are relevant with respect to t is obtained by a function $rel(bsh, t) : bSH^l \times Term \rightarrow bSH^l$ defined as:

$$rel(bsh, t) = \{s \mid s \in bsh, (s \bigwedge \hat{t}) \neq 0^l\}$$

where $\bigwedge$ represents the bitwise-and operation and 0^l is the all-zero string of length l. Consequently, the set of binary strings in $bsh \in bSH^l$ that are *irrelevant with respect to* t *is a function* $irrel(bsh, t) : bSH^l \times Term \rightarrow bSH^l$ *where* $irrel(bsh, t)$ *is the complement of* $rel(bsh, t)$*, i.e.,* $bsh \setminus rel(bsh, t)$.

Similarly, for two elements $r \in Term$ *and* $t \in Term$, $\hat{rt}$ *represents the binary representation of the union of the variables in* r *and* t.

Definition 5.3.3 (Binary cross-union, $\bowtie$). Given $bsh_1, bsh_2 \in bSH^l$, their *cross-union* is a function $\bowtie : bSH^l \times bSH^l \rightarrow bSH^l$ defined as

$$bsh_1 \bowtie bsh_2 = \{s \mid s = s_1 \bigvee s_2, s_1 \in bsh_1, s_2 \in bsh_2\}$$

where $\bigvee$ represents the bitwise-or operation.

Definition 5.3.4 (Binary up-closure, $(.)^*$). Let l be the length of strings in $bsh \in bSH^l$, then the *up-closure* of bsh, denoted bsh^* is a function $(.)^* : bSH^l \rightarrow bSH^l$ that represents the smallest superset of bsh such that $s_1 \bigvee s_2 \in bsh^*$ whenever $s_1, s_2 \in bsh^*$:

$$bsh^* = \{s \mid \exists n \geq 1 \; \exists t_1, \ldots, t_n \in bsh, \; s = t_1 \bigvee \ldots \bigvee t_n\}$$

Definition 5.3.5 (Binary abstract unification, $amgu$). The abstract unification is a function $amgu : \mathcal{V} \times Term \times bSH^l \rightarrow bSH^l$ defined as

$$amgu(x, t, bsh) = irrel(bsh, x = t) \; \cup \; (rel(bsh, x) \bowtie rel(bsh, t))^*$$

Example 5.3.1 (**Binary abstract unification**). Let $\mathcal{V} = \{X_1, X_2, X_3, X_4\}$ be the set of variables of interest and let $sh = \{\{X_1\}, \{X_2\}, \{X_3\}, \{X_4\}\}$ be a sharing set. Assume the following order among variables: $X_1 \prec X_2 \prec X_3 \prec X_4$. Then, we can easily encode each sharing group $sg \in sh$ into a binary string s such that $s[i] = 1$, $(1 \leq i \leq |sg|)$ if and only if the i-th variable of $\mathcal{V}$ appears in sg. In this example, sh is encoded as the following set of binary strings $bsh = \{1000, 0100, 0010, 0001\}$. Consider the analysis of $X_1 = f(X_2, X_3)$:

$$
\begin{array}{lcl}
A = rel(bsh, X_1) & = & \{1000\} \\
B = rel(bsh, f(X_2, X_3)) & = & \{0100, 0010\} \\
A \bowtie B & = & \{1100, 1010\} \\
(A \bowtie B)^* & = & \{1100, 1010, 1110\} \\
C = irrel(bsh, X_1 = f(X_2, X_3)) & = & \{0001\} \\
amgu(X_1, f(X_2, X_3), bsh) = C \ \cup \ (A \bowtie B)^* & = & \{0001, 1100, 1010, 1110\}
\end{array}
$$

The design of the analysis must be completed by defining the following abstract operations that are required by an analysis engine: *init* (initial abstract state), *equivalence* (between two abstract substitutions), *join* (defined as the union), and *project*.

Definition 5.3.6 (Binary projection, $bsh|_t$). The *binary projection* is a function $bsh|_t$: $bSH^l \times Term \rightarrow bSH^k$ $(k \leq l)$ that removes the i-th positions from all strings (of length l) in $bsh \in bSH^l$, if and only if the i-th positions of $\hat{t}$ (denoted by $\hat{t}[i]$) is 0, and it is defined as

$$bsh|_t = \{s' \mid s \in bsh, s' = \pi(s, t)\}$$

where $\pi(s, t)$ is the binary string projection defined as

$$\pi(s,t) = \begin{cases} \epsilon, & \text{if } s = \epsilon, \text{ the empty string} \\ \pi(s',t), & \text{if } s = s'a_i \text{ and } \hat{t}[i] = 0 \\ \pi(s',t)a_i, & \text{if } s = s'a_i \text{ and } \hat{t}[i] = 1 \end{cases}$$

and $s'a_i$ is the concatenation of character a to string s' at position i.

Example 5.3.2 Let $\mathcal{V}$ be the same set of variables of interest with the same order than in Example 5.3.1. Assume a set of sharing groups $sh = \{\{X_1, X_2\}, \{X_1, X_3\}, \{X_2, X_3\}, \{X_1, X_2, X_3\}, \{X_4\}\}$ encoded as $bsh = \{1100, 1010, 0110, 1110, 0001\}$. Then, the projection of bsh over the term $t = f(X_1, X_2, X_3)$ is $bsh|_t = \{110, 101, 011, 111\}$. Notice that since an all-zero string is meaningless in a set-sharing representation, it is dropped from the result.

Definition 5.3.7 (Binary initial state, *init*). The *initial state* is $init : \mathcal{V} \rightarrow bSH^{|\mathcal{V}|}$ describes an empty substitution resulting in $\{0^{|\mathcal{V}|}\}$.

Definition 5.3.8 (Binary equivalence, $=$). Given $bsh_1, bsh_2 \in bSH^l$, they are *equivalent* if and only if $(\forall s_1 \in bsh_1, \exists s_2 \in bsh_2, s_1 = s_2) \wedge (\forall s_2 \in bsh_2, \exists s_1 \in bsh_1, s_2 = s_1)$.

Definition 5.3.9 (Binary join, $\sqcup$). Given $bsh_1, bsh_2 \in bSH^l$, the *join* function $\sqcup : bSH^l \times bSH^l \rightarrow \wp(bSH^l)$ is defined as $bsh_1 \sqcup bsh_2 = bsh_1 \cup bsh_2$.

5.4 Ternary Set-Sharing

This section presents a more compact representation for the sharing domain defined above to accommodate a larger number of variables for analysis. We extend the binary string representation discussed above to use a ternary alphabet $\Sigma = \{0, 1, *\}$, where the $*$ symbol denotes both 0 *and* 1 bit values. This representation effectively compresses the number of elements in the set into fewer strings without changing the semantics of its representation. To handle the ternary alphabet, we redefine the binary operations covered in Section 5.3.

Definition 5.4.1 (Ternary Sharing Domain, tSH). Let alphabet $\Sigma_* = \{0, 1, *\}$, $\mathcal{V}$ be a fixed and finite set of variables of interest in an arbitrary order as in Def. 5.3.1, and Σ_*^l the finite set of all strings over Σ_* with length l, $0 \leq l \leq |\mathcal{V}|$. Then, $tSH^l = \wp^0(\Sigma_*^l)$ and hence, the *ternary sharing domain* is defined as $tSH = \bigcup_{0 \leq l \leq |\mathcal{V}|} tSH^l$.

Prior to defining how to transform the binary string representation into the corresponding ternary string representation, we introduce two core definitions, Definitions 5.4.2 and 5.4.3, for comparing ternary strings. These operations are essential for the conversion and ternary set operations. In addition, they are used to eliminate redundant strings within a set and may be used to check for equivalence of two ternary sets containing different strings.

Definition 5.4.2 (Match, $\mathcal{M}$). Given two ternary strings, $x, y \in \Sigma_*^l$, of length l, *match* is a function $\mathcal{M} : \Sigma_*^l \times \Sigma_*^l \rightarrow \mathcal{B}$, such that $\forall i\ 1 \leq i \leq l$,

$$x\ \mathcal{M}\ y = \begin{cases} \texttt{true}, \text{if } (x[i] = y[i]) \vee (x[i] = *) \vee (y[i] = *) \\ \texttt{false}, \text{otherwise} \end{cases}$$

Definition 5.4.3 (Subsumed_By $\trianglelefteq$ and Subsumed_In $\trianglelefteqq$). Given two ternary strings $s_1, s_2 \in \Sigma_*^l$, $\trianglelefteq : \Sigma_*^l \times \Sigma_*^l \rightarrow \mathcal{B}$ is a function such that $s_1 \trianglelefteq s_2$ if and only if every string matched by s_1 is also matched by s_2. More formally, $s_1 \trianglelefteq s_2 \Longleftrightarrow \forall s \in tSH^l,\ if\ s_1\ \mathcal{M}\ s\ \ then\ \ s_2\ \mathcal{M}\ s$. For convenience, we augment this definition to deal with sets of strings. Given a ternary string $s \in \Sigma_*^l$ and a ternary sharing set, $tsh \in tSH^l$, $\trianglelefteqq : \Sigma_*^l \times tSH^l \rightarrow \mathcal{B}$ is a function such that $s \trianglelefteqq tsh$ if and only if there exists some element $s' \in tsh$ such that $s \trianglelefteq s'$.

One important distinction between the two definitions above is that two strings may match but neither may subsume the other. For example, 1*1 $\mathcal{M}$ *01, but 1*1 $\not\trianglelefteq$ *01 and 0*1 $\not\trianglelefteq$ 1*1. Match and subsumption checks between two strings of length l takes $O(l)$. Verifying if a string is subsumed by some string in a set tsh takes $O(|tsh|l)$.

The original NDB concept presented in [41] produced a set of records (or strings) representing the input in a random, obfuscated fashion such that it is difficult to extract information from the database. Thus, negative pattern generate and supporting operations were developed to maximize obscurity of a negative set of records. Redundancy among the records was accepted at the expense of the resulting set being far from minimal in size.

Figure 5.1 gives the pseudo code for an algorithm which converts a set of binary strings into a set of compact, equivalent ternary strings. The function `Convert` evaluates each input string and attempts to introduce $*$ symbols using `PatternGenerate` based on the k value, while eliminating redundant strings using `ManagedGrowth`. The overall time complexity of `Convert` is $O(|bsh|\alpha l)$, where bsh is the input set of l-length binary strings and α equals to the maximum size of the intermediate result times l. The largest size the resulting set will grow is the size of the binary set, $O(|bsh|)$.

```
0  Convert(bsh, k)
1  tsh ← ∅
2  foreach s ∈ bsh
3      y ← PatternGenerate(tsh, s, k)
4      tsh ← ManagedGrowth(tsh, y)
5  return tsh
```

```
10 PatternGenerate(tsh, x, k)
11 m ← Specified(x)
12 i ← 0
13 x' ← x
14 l ← length(x)
15 while m > k and i < l
16     Let b_i be the value of x' at position i
17     if b_i = 0 or b_i = 1 then
18         x' ← x' with position i replaced by ¬b_i
19         if x' ⊑ tsh then
20             x' ← x' with position i replaced by ∗
21         else
22             x' ← x' with position i replaced by b_i
23     m ← Specified(x')
24     i ← i + 1
25 return x'
```

```
30 ManagedGrowth(tsh, y)
31 S_y = {s | s ∈ tsh, s ⊑ y}
32 if S_y = ∅ then
33     if y ⋢ tsh then
34         append y to tsh
35 else
36     remove S_y from tsh
37     append y to tsh
38 return tsh
```

Figure 5.1: A deterministic algorithm for converting a set of binary strings bsh into a set of ternary strings tsh, where k is the desired minimum number of specified bits (non-$*$) to remain.

5.4.1 Pattern Generate

This version of pattern generate removes all randomization and deterministically create ternary strings (or records). `PatternGenerate`, see Figure 5.1, begins by evaluating the input string bit-by-bit to determine where the $*$ symbol can be introduced. `PatternGenerate` was adapted from the randomized version defined by Esponda in [41]. In our version, since no privacy or security properties are required, all randomizations were eliminated. The algorithm iterates while the number of specified bits are greater than the desired minimum k (and the end of the string has not been reached). The number of $*$ symbols introduced depends on the sharing

set represented and k, the desired minimum number of specified bits, $0 \leq k \leq l$ (the string length). Thus, for a given set of strings of length l, parameter k controls the compression of the set. For $k = l$ (all bits specified), there is no compression and $tsh = bsh$. For a non-empty bsh, $k = 1$ introduces the maximum number of $*$ symbols. For now, we will assume that $k = 1$, and some experimental results shown by Figure 5.7 will show the best overall k value for a given l is near $l/2$.

The `Specified` function returns the number of specified bits (0 or 1) in x. At each step, if the i-th bit (b_i) is specified, then a new string is created with the original values, except with b_i is *substituted* ($\cdot$ symbol) by the opposite value, $\overline{b_i}$. If this new string is subsumed by tsh, then a $*$ can be substituted at b_i. Otherwise, b_i is reset to its original value. The new string x' with more $*$'s is returned. `PatternGenerate` has a worst case time complexity for input tsh with strings of length l of $O(\alpha l)$, where $\alpha = |tsh| \cdot l$.

5.4.2 Manage Growth

To better control the size of the resulting sharing sets, it is crucial to disallow duplicates or redundant strings when adding strings to the sharing set. `ManagedGrowth`, see Figure 5.1, ensures that the candidate string y is not redundant to existing ones in the set tsh, and it also checks if y subsumes other strings in tsh. In other words, if no redundant string exists, then y is appended to tsh only if y itself is not redundant to an existing string in tsh. Otherwise, y replaces all the redundant strings in tsh.

Employing `ManagedGrowth` decreases the size of intermediate working sets and the final result set. For example, as shown by Table 5.4.1, after adding the second record 001 with managed growth resulted in only two records (1**, *1*). However, some partial redundancy remains to maintain the semantic correctness of the representation. From the example, the both strings, 1** and *1*, represent the string

Input	*Unmanaged*	*Managed*	$\Delta Size$
000	1** *1* **1	1** *1* **1	0
001	1** *1* 1*1 *11	1** *1*	-2
010	1** *11 1*1 11*	1** *11	-2
011	1** 1*1 11* 111	1**	-3
100	1*1 11* 111	1*1 11*	-1
101	11* 111	11*	-1
110	111	111	0
111			0

Table 5.1: Set size is reduced by not allowing redundant, equivalent strings. The Unmanaged column shows the number of strings without controlling the growth as compared to the Managed column (size differences shown in ΔSize column).

111. Removing either one will result in an incorrect representation. This is more significantly illustrated in Figure 5.7 where $k = 1$. In this case with $l = 12$, many partial redundancies exists and results in output sizes larger than 2^l. `ManagedGrowth` has a worst case time complexity for input tsh with strings of length l of $O(\alpha)$, where $\alpha = |tsh| \cdot l$.

Figure 5.2 illustrates the result of using *Convert* to transform 2^8 binary to ternary strings, in random order, with $k = 4$ minimum specified bits (for 30 experiments). The graph shows that the size of the ternary representation is consistently less than

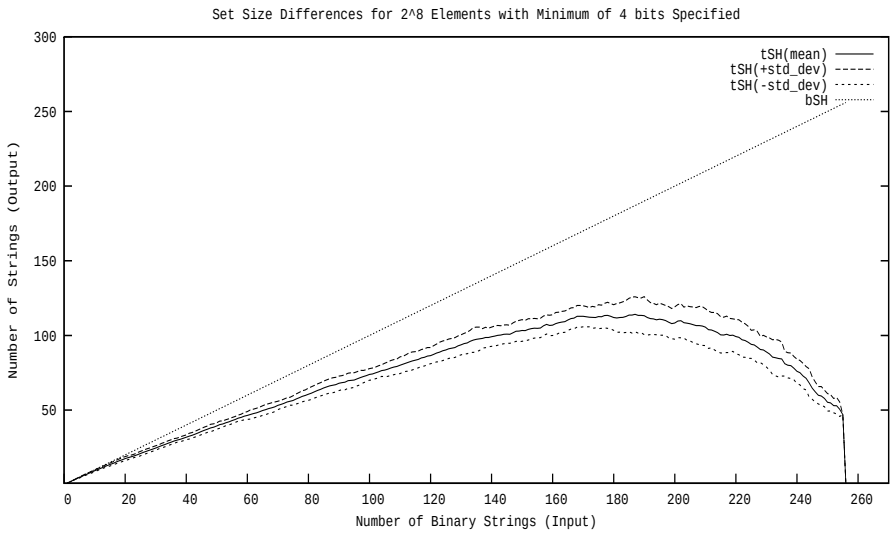

Figure 5.2: Average size comparisons of binary (bSH) and ternary (tSH) string formats.

the binary. This size difference helps attain favorable results with respect to result size and completion time of abstract unification, see Section 5.6.

5.4.3 Example of Conversion from bSH to tSH

Example 5.4.1 (**Conversion from bSH to tSH**). Let $\mathcal{V}$ be the set of variables of interest with the same order as Example 5.3.1. Assume the following sharing set of binary strings $bsh = \{1000, 1001, 0100, 0101, 0010, 0001\}$. Then, a ternary string representation produced by applying `Convert` is $tsh = \{100*, 0010, 010*, *001\}$. Notice there remains a certain level of redundancy in the representation, a subject that will be discussed further in Section 5.6.

The example above begins with `Convert`(bsh,$k = 1$). Since $tsh = \emptyset$ initially

(line 1), the first string 1000 is appended to tsh, so $tsh = \{1000\}$. Next, 1001 from bsh is evaluated. In `PatternGenerate`, with x' at iteration i (denoted as x'_i), $i = 3$, and $b_3 = 1$ ($x'_3 = 1000$), we test if the i^{th} position of x can be replaced with a $*$ (line 15-24). In this case, since $x'_3 \trianglelefteqq tsh$ (line 19), $x'_3 = 100*$ is returned (line 25). Next, `ManagedGrowth` evaluates 100* and since it subsumes 1000 ($S_y = \{1000\}$), 100* replaces 1000 leaving $tsh = \{100*\}$ (line 38). The process continues with `PatternGenerate`({100*},0100) (line 3). In `PatternGenerate`, since $x'_0 \not\trianglelefteqq tsh$, $x'_1 \not\trianglelefteqq tsh$, $x'_2 \not\trianglelefteqq tsh$, and $x'_3 \not\trianglelefteqq tsh$, we reset each i^{th} bit to its original value (line 22) and $x' = x = 0100$ is returned. Next, `ManagedGrowth`({100*},0100) is called and since 0100 is not redundant to any string in tsh, it is simply appended resulting in $tsh =$ {100*,0100}. The process continues with `PatternGenerate`({100*,0100},0101). In `PatternGenerate`, when $x'_3 = 0100$ and since $x'_3 \trianglelefteqq tsh$, then $x'_3 = 010*$ is returned. `ManagedGrowth`({100*, 0100}, 010*) is called next and since 010* subsumes 0100 $\in tsh$, 0100 is replaced resulting in $tsh =$ {100*,010*} (line 38). The process continues similarly, for the remaining input strings in bsh obtaining the final result of $tsh =$ {100*, 0010, 010*, *001}.

A useful standalone operation to further decrease the size of the ternary set is

```
0  Compress(sh, k, c)
1  repeat c times
2      sh' ← sh
3      foreach s ∈ sh'
4          sh'' ← sh' \ {s}
5          y ← PatternGenerate(sh'', s, k)
6          if y ⋤ sh'' then
7              append y to sh''
8          sh' ← sh''
9      if |sh| = |sh'| then
10         return sh'
11 return sh
```

Figure 5.3: `Compress` algorithm to remove redundancies in a sharing set.

`Compress`, which eliminates redundant strings in the set. It is useful after abstract unification operation or as warranted. `Compress`, as shown in Figure 5.3, iterates through the set and attempts to add $*$ symbols (using a low k value) for each string in the set. It uses `PatternGenerate` to introduce $*$ symbols while eliminating redundant strings using `ManagedGrowth` . The process continues for some constant c times or until no improvement (decrease in size) from the previous iteration occurs. `Compress` has a worst case time complexity, for input sharing set sh containing l-length strings, of $O(\alpha|sh|)$, where $\alpha = |sh| \cdot l$.

5.4.4 Ternary Set-Sharing Operations

Each of the binary string operations from Section 5.3 is redefined to account for the $*$ symbol in a ternary string. Note that since the ternary representation extends the binary alphabet (i.e., binary is a subset of the ternary alphabet), ternary operations can also operate over strictly binary strings. For simplicity, certain operators are overloaded to denote operations involving both binary and ternary strings.

Definition 5.4.4 (Ternary-or $\bigvee$ and Ternary-and $\bigwedge$). Given two ternary strings, $x, y \in \Sigma_*^l$ of length l, *ternary-or* and *ternary-and* are two bitwise functions defined as $\bigvee, \bigwedge : \Sigma_*^l \times \Sigma_*^l \to \Sigma_*^l$ such that $z = x \bigvee y$ and $w = x \bigwedge y$, $\forall i\ 1 \leq i \leq l$, where:

$$z[i] = \begin{cases} * & \text{if } (x[i] = * \wedge y[i] = *) \\ 0 & \text{if } (x[i] = 0 \wedge y[i] = 0) \\ 1 & \textit{otherwise} \end{cases} \qquad w[i] = \begin{cases} * & \text{if } (x[i] = * \wedge y[i] = *) \\ 1 & \text{if } (x[i] = 1 \wedge y[i] = 1) \\ & \vee\ (x[i] = 1 \wedge y[i] = *) \\ & \vee\ (x[i] = * \wedge y[i] = 1) \\ 0 & \textit{otherwise} \end{cases}$$

Definition 5.4.5 (Ternary set intersection, $\cap$). Given tsh_1, $tsh_2 \in tSH^l$, $\cap$:

$tSH^l \times tSH^l \rightarrow tSH^l$ is defined as

$$tsh_1 \ \cap \ tsh_2 = \{r \mid r = s1 \bigwedge s2, s1 \ \mathcal{M} \ s2, s1 \in tsh_1, s2 \in tsh_2\}$$

For convenience, we define two binary patterns, `0-mask` and `1-mask`, in order to simplify operations defined below. The former takes an l-length binary string s and returns a set with a single string having a 0 where $s[i] = 1$ and $*$'s elsewhere, $\forall i \ 1 \leq i \leq l$. The latter takes also an l-length binary string s, but returns a set of strings with a 1 where $s[i] = 1$ and $*$'s elsewhere, $\forall i \ 1 \leq i \leq l$. For instance, `0-mask`(0110) and `1-mask`(0110) return $\{*00*\}$ and $\{*1**, **1*\}$, respectively.

Definition 5.4.6 (Ternary relevant sharing $rel(tsh, t)$ and irrelevant sharing $irrel(tsh, t)$). Given $t \in Term$ with length l and $tsh \in tSH^l$ with strings of length l, the set of strings in tsh that are *relevant* with respect to t is obtained by a function $rel(tsh, t) : tSH^l \times Term \rightarrow tSH^l$ defined as

$$rel(tsh, t) = tsh \cap \texttt{1-mask}(\hat{t})$$

In addition, $irrel(tsh, t)$ is defined as

$$irrel(tsh, t) = (tsh \ \cap \ \texttt{1-mask}(\bar{\hat{t}})) \cap \texttt{0-mask}(\hat{t})$$

Ternary cross-union, $\bowtie$, and ternary up-closure, $(.)^*$, operations are as defined in Definition 5.3.3 and in Definition 5.3.4, respectively, except the binary version of the bitwise-or operator is replaced with its ternary counterpart defined in Definition 5.4.4 in order to account for the $*$ symbol. In addition, the ternary abstract unification ($amgu$) is defined exactly as the binary version, Definition 5.3.5, using the corresponding ternary definitions.

Example 5.4.2 (**Ternary abstract unification**). Let tsh = {100*, 010*, 0010, *001} as in Example 5.4.1. Consider again the analysis of $X_1 = f(X_2, X_3)$, the result is:

$$\begin{array}{lcl} A = rel(tsh, X_1) & = & \{100*\} \\ B = rel(tsh, f(X_2, X_3)) & = & \{010*, 0010\} \\ A \bowtie B & = & \{110*, 101*\} \\ (A \bowtie B)^* & = & \{110*, 101*, 111*\} \\ C = irrel(tsh, X_1 = f(X_2, X_3)) & = & \{0001\} \\ amgu(X_1, f(X_2, X_3), tsh) = C \ \cup \ (A \bowtie B)^* & = & \{0001, 110*, 101*, 111*\} \end{array}$$

Ternary projection, $tsh|_t$, is defined similarly as binary projection, see Def. 5.3.6. However, the projection domain and range is extended to accommodate the $*$ symbol. So, the function definition remains the same except that *ternary* string projection is now defined as a function $\pi(s, t)$: $\Sigma_*^l \times Term \rightarrow \Sigma_*^k$, $k \leq l$. For example, let tsh = {100*, 010*, 0010, *001} as in Example 5.4.1. Then, the projection of tsh over the term $t = f(X_1, X_2, X_3)$ is $tsh|_t$ = {100, 010, 001}. Once again, note that since an all-zero string is meaningless in a set-sharing representation, it is not included here.

Definition 5.4.7 (Ternary initial state, *init*). The *initial state* $init : \mathcal{V} \times \mathcal{I}^+ \rightarrow tSH^{|\mathcal{V}|}$ describes an initial substitution given a set of variables of interest. Assuming the binary initial state operation defined as $init_{bSH} : \mathcal{V} \rightarrow bSH^{|\mathcal{V}|}$, the ternary initial state can be defined using the `Convert` algorithm in Fig. 5.1 as:

$$init(\mathcal{V}, k) = \texttt{Convert}(init_{bSH}(\mathcal{V}), k)$$

Definition 5.4.8 (Ternary equivalence, $\equiv$). Given tsh_1, $tsh_2 \in tSH^l$, the sets are *equivalent* if and only $(\forall t_1 \in tsh_1, \forall s_1 \subseteq t_1, \ s_1 \subseteq tsh_2) \ \wedge \ (\forall t_2 \in tsh_2, \forall s_2 \subseteq t_2,$

$s_2 \trianglelefteqq tsh_1$). Note that two ternary sets may represent the same sharing set even though they *contain* different ternary strings.

To compare two ternary sets for equivalence, we cannot simply check for equality of elements of the sets. Two ternary sets *containing* different strings may represent the same sharing set, e.g., $\{101, *11\} \equiv \{1*1, *11\}$. Therefore, each strings subsumed by all strings in one set must be represented in the other. In the worst case, this may involved resolving all $*$ symbols and checking for equality, leading to $O(2^{l-k})$ operation, with strings of length l and k specified bits.

Finally, the ternary join is defined as its binary counterpart, i.e., set union.

Definition 5.4.9 (Ternary join, $\sqcup$). Given $tsh_1, tsh_2 \in tSH$, the *join* function $\sqcup : tSH \times tSH \rightarrow \wp(tSH)$ is defined as the set union of the two sets, $tsh_1 \sqcup tsh_2 = tsh_1 \cup tsh_2$.

This section presented an extension to the binary string representation using a ternary alphabet. This extension effectively compressed any set-sharing problem instance allowing completion of abstract unification over larger number of variables of interest, see Section 5.6. In addition, a conversion algorithm was introduced and associated operators were redefined to accommodate the new representation.

5.5 Negative Ternary Set-Sharing

In this section, further enhancement to the ternary representation presented in the previous section is described. The main idea is that in certain cases, a more compact representation of sharing relationships among variables can be captured equivalently

by working with the complement (or negative) set of the original sharing set. A ternary string t can either be *in* or *not in* the set $tsh \in tSH$. This mutual exclusivity together with the finiteness of $\mathcal{V}$ allows for checking t's membership in tsh by asking if t is in tsh, or, equivalently, if t is *not* in its complement, $\overline{tsh}$. Given a set of l-bit binary strings, its complement or negative set contains *all* the l-bit ternary strings *not* in the original set. Therefore, if the cardinality of a set is greater than half of the maximum size, i.e., $> 2^{|\mathcal{V}|-1}$, then the size of its complement will be less than $2^{|\mathcal{V}|-1}$. It is this size differential that we exploit. In Set-Sharing analysis, as we consider programs with larger numbers of variables of interest, the potential number of sharing groups grows exponentially, toward $2^{|\mathcal{V}|}$, whereas the number of sharing groups *not* in the sharing set decreases toward 0.

The idea of a negative set representation and its associated algorithms extends the work by Esponda et al. in [41, 46]. In that work, a negative set is generated from the original set in a similar manner to the conversion algorithms shown in Figures 5.1 and 5.4. However, they produce a negative set with unspecified bits in random positions and with less emphasis on managing the growth of the resulting set. The technique was originally introduced as a means of generating Boolean satisfiability (SAT) formulas where, by leveraging the difficulty of finding solutions to hard SAT instances, the contents of the original set are obscured without using encryption [41]. In addition, these hard-to-reverse negative sets are still able to answer membership queries efficiently while remaining intractable to reverse (i.e., to obtain the contents of the original set). In this application, this important security property is not required, however, and it uses the negative approach simply to address the efficiency issues faced by the traditional Set-Sharing domain.

The conversion to the negative set can be accomplished using the two algorithms shown in Figure 5.4. These algorithms consist of two operations that insert and delete strings from a ternary set. `Insert` adds ternary strings with a number of

specified bits based on the k parameter. If the number of specified bits m for a given string x is less than k, then $k - m$ unspecified bits are selected and all 2^{k-m} specified string combinations are added to the result (filtered for redundancies by `ManagedGrowth`). Otherwise $m \geq k$, there are more specified bits than k and the algorithm attempts to compress the set by introducing $*$'s using `PatternGenerate` and filtered by `ManagedGrowth`. Example 5.5.1 illustrates this process with a concrete example. The worst case time complexity for `Insert` is $O(2^{\delta} \cdot$ O(`ManagedGrowth`)), where $\delta = k - m$ if $m < k$, 0 otherwise. So, `Insert` takes $O(2^{\delta} \cdot \alpha)$, where $\alpha = |tnsh| \cdot l$. The worst case space complexity of `Insert` is $O(2^{\delta}|tnsh|)$.

The `Delete` operation removes all strings matching x and saves it in a set D_x. It also uses `Insert` to re-add strings that were matched by x but not subsumed by x. This step is critical in maintaining the correctness of the representation since strings in this "deleted cache" D_x may represent other strings not represented by x. For each bit location specified in x and not specified in string $y \in D_x$, `Delete` creates candidate string y' by setting that bit location to the opposite bit value specified in x. Then, each y' is added into the result set using `Insert`. The worst case time complexity for `Delete` is $O(|tnsh|l + (|tnsh|l \cdot$ O(`Insert`)) $= O(\alpha(\alpha 2^{\delta} + 1))$, where $\alpha = |tnsh| \cdot l$. The worst case space complexity for `Delete` is $O(2^{\delta}(l - m))$, where m is the smallest number of specified bits in the entire set.

To convert a positive (binary or ternary) set sh to its negative (complement) representation, two options are provided. First, `NegConvert` uses the `Delete` operation to remove postive strings in the input set sh from $\mathcal{U}$, the set of all l-bit strings $\mathcal{U} = \{*^l\}$. Then, it calls `Insert` to return $\mathcal{U} \setminus sh$ which results in all strings *not* in the original input—its complement. The worst case space and time complexity for `NegConvert` is based on whether the conversion is from positive to negative or negative to positive and the input size times O(`Delete`), see Table 5.2.

Alternatively, `NegConvertMissing` may be used to convert a positive set to its

negative representation. `NegConvertMissing` initially computes the *missing* strings from the input. Then, each missing string is inserted into an initially empty result set. This conversion results in a negative set representing all strings *not* in the original input. The overall time complexity of `NegConvertMissing` depends on the time it takes to find all the missing strings, denoted as β, plus the number of missing strings $|bnsh|$ times O(`Insert`) resulting in $O(\beta + |bnsh|(\alpha 2^{\delta}))$ (see Table 5.2). The worst case space complexity for `NegConvertMissing` depends on the number of missing strings computed times new strings created resulting in $O(|bnsh|2^{\delta})$.

Table 5.2 shows that both algorithms have slightly different complexities. The more efficient conversion algorithm depends on the size of the input. For small number of variables, it may be more advantageous to find all the strings missing from the input by iterating through the entire power set (a low β value) and then transforming them using `NegConvertMissing`. However, for larger positive sets (smaller negative sets) with longer string lengths, it may be more efficient to use `NegConvert`.

As hinted above, `NegConvert` can be used to efficiently convert a negative set to its complement—its positive ternary representation. This functionality has enabled, in most cases, a more efficient computation of cross-union in the negative representation. Cross-union, an $\mathcal{NP}$-Hard operation (see Theorem 9), required establishing all positive string represented by reversing the negative set. To recover the positive binary representation from the negative set requires exponential time, $O(2^{l-m})$. However, by converting to the ternary representation, fully resolving to the binary representation is avoided resulting in a more efficient abstract unification, see results in Figure 5.9. However, in the worst case, as shown by Table 5.2, the conversion process remains exponential.

Note that the resulting negative set uses the same ternary alphabet described in Definition 5.4.1 for the ternary sharing domain. For clarity, the negative ternary set-sharing domain is denoted as $tNSH$ even though $tNSH \equiv tSH$.

```
0  NegConvert(sh, k)
1  tnsh ← 𝒰
2  foreach t ∈ sh
3      tnsh ← Delete(tnsh, t, k)
4  return tnsh
```

```
0  NegConvertMissing(bsh, k)
1  tnsh ← ∅
2  bnsh ← 𝒰 \ bsh
3  foreach t ∈ bnsh
4      tnsh ← Insert(tnsh, t, k)
5  return tnsh
```

```
10 Delete(tnsh, x, k)
11 D_x ← ∀t ∈ tnsh, x ℳ t
12 tnsh' ← tnsh with D_x removed
13 foreach y ∈ D_x
14     foreach unspecified bit position q_i of y
15         if b_i (the i^th bit of x) is specified, then
16             y' ← y with position q_i replaced by b̄_i
17             tnsh' ← Insert(tnsh', y', k)
18 return tnsh'
```

```
20 Insert(tnsh, x, k)
21 m ← Specified(x)
22 if m < k then
23     P ← select (k − m) unspecified bit positions in x
24     V_P ← every possible bit assignment of length |P|
25     foreach v ∈ V_P
26         y ← x with positions P replaced by v
27         tnsh' ← ManagedGrowth(tnsh, y)
28 else
29     y ← PatternGenerate(tnsh, x, k)
30     tnsh' ← ManagedGrowth(tnsh, y)
31 return tnsh'
```

Figure 5.4: `NegConvert`, `NegConvertMissing`, `Delete` and `Insert` algorithms used to transform positive to negative representation; k is the desired number of specified bits (non-*'s) to remain.

5.5.1 Deleted Cache Size

With regards to managing the growth of the negative set, studying the deleted cache size helps determine the potential number of new records that will be created when adding a new positive string (which must be deleted from the negative set). The number of unspecified bits, $*$, establishes the potential number of strings to be added

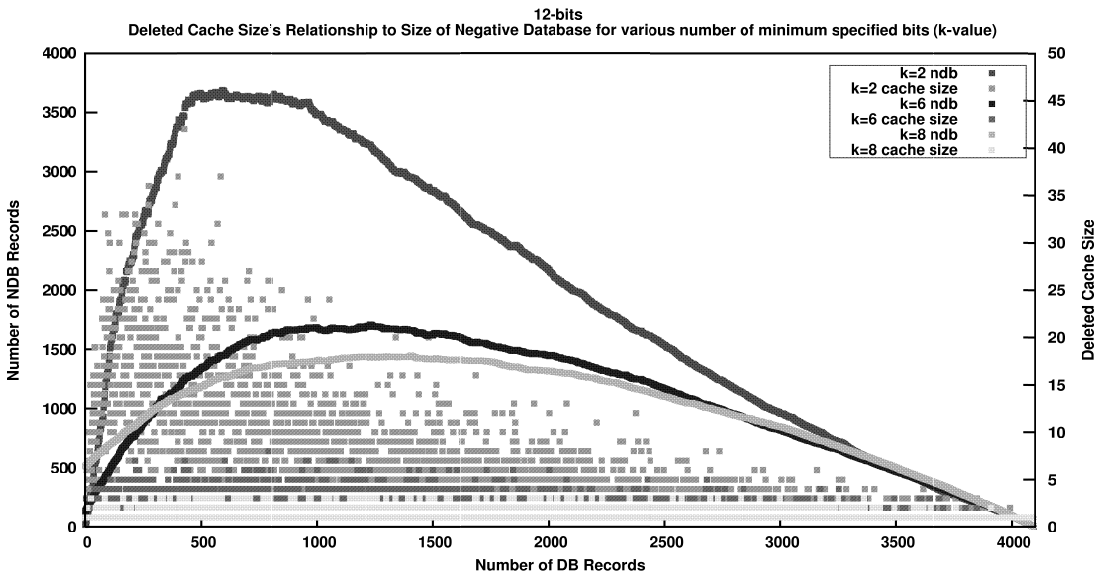

Figure 5.5: Size of resulting sets (left label) increases as the number of $*$ symbols are introduced due to increased number of candidate records, shown by the Deleted cache size (right label), matching the string to be removed. Note: k determines minimum number of specified bits per record; larger k results in less $*$'s in each record.

as shown in the `Delete` algorithm in Figure 5.4. The deleted cache size, D_x, is large when there exists many $*$'s in the negative set. Each $*$ symbol in each matched record in the cache generates an additional new record in the resulting set. Figure 5.5 illustrates that as the number specified bits increase, the number of negative strings decrease. However, the minimum number of specified bits, k, should not equal the length of each record because no compression would take place. Thus, there is a trade off between maximizing the compression level of the current working set and the potential for a large increase in subsequent working sets.

Through several experimentation with different string lengths and k value, the best range of values for k is $l/2 \leq k < l$, see Figure 5.7.

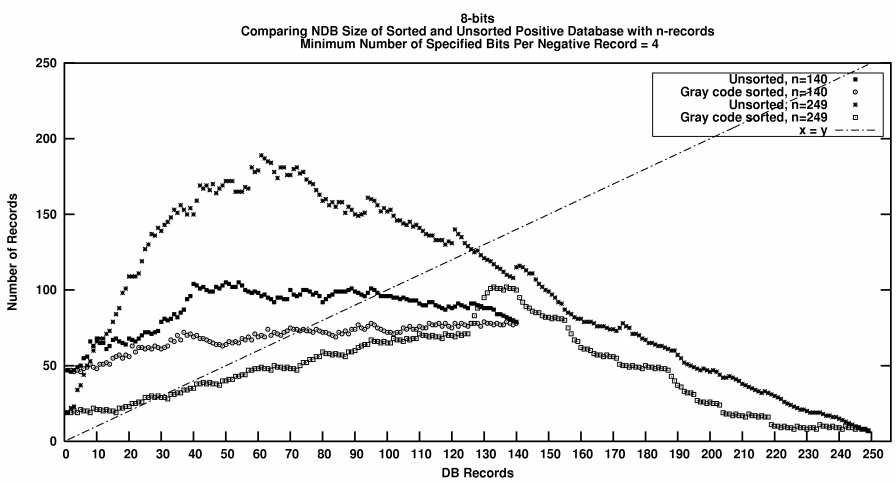

Figure 5.6: Size comparisons with Sorted and Unsorted input

5.5.2 Sorting

An important research result illustrates how sorting the input binary set dramatically reduces the number of elements in the set during conversions. Inserting strings that are a small Hamming distance away from previous strings reduces the intermediate size of the result set. Since our algorithms are deterministic, the size of the final result set depends primarily on the input to be converted. Although the final result is the same for the same input set (sorted or not), the size of the intermediate working sets are significantly reduced when the input is sorted. Figure 5.6 shows that as the size of the input grows, sorting does improve the intermediate working set even though both unsorted and sorted versions converge to the same result set.

We discovered that different types of sorting produce varying improvements. For bSH, sorting numerically by binary value results in a smaller size than an unsorted input. However, the Hamming distance at points where the binary bits carry to the next significant digit occurs result in less than optimum conversion. For example,

strings 011 and 100 has a Hamming distance of 3, whereas 011 and 010 only has a distance of 1. Two strings with a distance of one can be compressed onto a single string by replacing the bits where they differ by a $*$. The best compression was achieved by sorting using Gray codes [97]. Gray coding sorts strings such that adjacent strings have the minimum Hamming distance possible, without regard to binary value or alphanumeric ordering.

5.5.3 Example of Conversion from bSH to tNSH

Example 5.5.1 (**Conversion from bSH to tNSH**). Consider the same sharing set as in Example 5.4.1: bsh = {1000, 1001, 0100, 0010, 0101, 0001}. A negative ternary string representation is generated by applying the `NegConvert` algorithm to obtain {0000, 11**, 1*1*, *11*, **11}. Since a string of all 0's is meaningless in a set-sharing representation, it is removed from the set. Thus, $tnsh$ = {11**, 1*1*, *11*, **11}.

For Example 5.5.1, the first string 1000 is deleted from $\mathcal{U} = \{****\}$. So, $D_x = \{****\}$ (line 11) and $tnsh' = \emptyset$ (line 12). For each i^{th} bit of x, a new $y'_i \not\asymp x$ is evaluated for insertion into the result set. So, `Insert` ($\emptyset$, $y'_0 = 0***$, $k = 1$) is called (line 17). Since `Specified`$(y') \geq k$ and $tnsh' = \emptyset$, the result returned is $tnsh'$ ={0***} (line 27-30). For all other unspecified positions (line 14) of y, a new string is created with a bit value opposite to x_i's value, ($\overline{b_i}$). So, `Insert` ({0***}, $y'_1 = *1**$, $k = 1$) is called next and y'_1 is appended to $tnsh'$. The process continues with y'_2 and y'_3 resulting in $tnsh$ = {0***, *1**, **1*, ***1}.

Next, 1001 from bsh is deleted (line 2) resulting in D_x ={***1} and $tnsh'$ = {0***, *1**, **1*} (line 11,12). Then, `Insert` ({0***, *1**, **1*}, $y' = 0**1$, $k = 1$) is called. Since $0**1 \sqsubseteq tnsh'$, then $tnsh'$ remains unchanged. The process continues with y'_1 =*1*1, y'_2 =**11 being subsumed by $tnsh'$; so the result returned

is $tnsh$ = {0***, *1**, **1*}. Next, 0100 is deleted resulting in $tnsh$ = {00**, 0**1, 11**, *1*1, **1*}. Next, 0010 is deleted resulting in $tnsh$ = {000*, 0**1, 11**, 1*1*, *11*, *1*1, **11}. Next, 0101 is deleted resulting in $tnsh$ = {000*, 00*1, 11**, 1*1*, *11*, **11}. Finally, 0001 is deleted resulting in $tnsh$ = {0000, 11**, 1*1*, *11*, **11}. Removing the string with all 0s, we get the final $tnsh$ = {11**, 1*1*, *11*, **11}. Notice that $tnsh = \mathcal{U} \setminus (bsh \cup \{0000\})$.

Alternatively, `NegConvertMissing` may be used to convert in the following way. first the missing strings must be calculated from the given set. For Example 5.5.1, the missing strings are {0011, 0110, 0111, 1010, 1011, 1100, 1101, 1110, 1111}. The `NegConvertMissing` begins with the first string 0011 and $tnsh = \emptyset$ resulting in $tnsh$ ={0011}.

Then, `Insert` ({0011}, $y' = 0110$, $k = 1$) resulting in $tnsh$ ={0011, 0110}. Next, `Insert` ({0011, 0110}, $y' = 0111$, $k = 1$) resulting in $tnsh$ ={011*, 0*11}. Next, `Insert` ({011*, 0*11}, $y' = 1010$, $k = 1$) resulting in $tnsh$ ={011*, 0*11, 1010}. Next, `Insert` ({011*, 0*11, 1010}, $y' = 1011$, $k = 1$) resulting in $tnsh$ ={011*, 0*11, 101*, *011}. Next, `Insert` ({011*, 0*11, 101*, *011}, $y' = 1100$, $k = 1$) resulting in $tnsh$ ={011*, 0*11, 101*, 1100, *011}. Next, `Insert` ({011*, 0*11, 101*, 1100, *011}, $y' = 1101$, $k = 1$) resulting in $tnsh$ ={011*, 0*11, 101*, 110*, *011}. Next, `Insert` ({011*, 0*11, 101*, 110*, *011}, $y' = 1110$, $k = 1$) resulting in $tnsh$ ={011*, 0*11, 101*, 110*, *011, *110}. Finally, `Insert` ({011*, 0*11, 101*, 110*, *011, *110}, $y' = 1111$, $k = 1$) resulting in $tnsh$ ={11**, 1*1*, *11*, **11}.

Notice that the final result of both conversion algorithms are equivalent. Table 5.2 illustrates different transformations and their results for a given input and convert operation. Rows 2 and 4 show that both `NegConvert` and `NegConvertMissing` can convert a positive representation into its corresponding negative set. Depending on the size of the original input we may prefer one transformation over another. If the input size is relatively small, less than 50% of the maximum size, then `NegConvert`

Input	Convert Operation	Result	Description	Time Complexity	Size Complexity
bsh	`Convert`	tsh	bSH to tSH	$O(\|bsh\|\alpha l)$	$O(\|bsh\|)$
bsh/tsh	`NegConvert`	$tnsh$	pos to neg	$O(\|bsh\|\alpha(\alpha 2^{\delta}+1))$	$O(\|tnsh\|(l-m)2^{\delta})$
$tnsh$	`NegConvert`	tsh	neg to pos	$O(\|tnsh\|\alpha(\alpha 2^{\delta}+1))$	$O(\|tsh\|(l-m)2^{\delta})$
bsh	`NegConvertMissing`	$tnsh$	pos to neg	$O(\beta+\|bnsh\|(\alpha 2^{\delta}))$	$O(\|bnsh\|2^{\delta})$

Table 5.2: Summary of conversions: l-length strings; $\alpha = |Result| \cdot l$; if $m < k$ then $\delta = k - m$ else $\delta = 0$, where $m =$ minimum specified bits in entire set, $k =$ number of specified bits desired; $bnsh = \mathcal{U} \setminus bsh$; $\beta = O(2^l)$ time to find $bnsh$.

is often more efficient than `NegConvertMissing`. Alternatively, we may initially find the missing strings from the input set and incrementally `Insert` each one into an empty negative set.

Consider the same set of variables and order among them as in Example 5.5.1 but with a slightly different set of sharing groups encoded as bsh = {1000, 1100, 1110} or tsh = {1*00, 1110}. The negative ternary string representation produced by `NegConvert` is $tnsh$ ={00**, 01**, 0*1*, 0**1, 1**1, *01*}. This example shows that the number of elements, or size, of the negative result, $|tnsh| = 6 > |bsh| = 3$ and $|tsh| = 2$. However, in Example 5.5.1 when $|bsh| = 6$, $|tnsh| = 4 < |bsh|$. This is because when $|bsh|$ is less than $2^{|\mathcal{V}|-1}$, i.e., $|bsh| = 3 < 2^3$, then its complement set must represent $(2^{|\mathcal{V}|} - |bsh|) = 13$ elements. Depending on the strings in the positive set, the size of the negative result may indeed be greater. This is a good illustration of how selecting the appropriate set-sharing representation will affect the size of the converted result. Thus, the size of the original sharing set at specific program points will be used by the analysis to produce the most compact working set. In Section 5.6, we show that the smaller the size of the representation, the faster *amgu* completes execution and the smaller the resulting set generated. The negative sharing set representation allows us to represent more variables of interest enabling larger problem instances to be evaluated.

5.5.4 Negative Ternary Set-Sharing Operations

Operations are now defined for negative ternary representation in order to perform abstract unification and the other abstract operations required by our engine to use the negative representation.

Definition 5.5.1 (Negative relevant sharing $\overline{rel}(tnsh, t)$ and irrelevant sharing $\overline{irrel}(tnsh, t)$) Given $t \in Term$ and $tnsh \in tNSH^l$ with strings of length l, the set of strings in $tnsh$ that are *negative relevant* with respect to t is obtained by a function $\overline{rel}(tnsh, t) : tNSH^l \times Term \rightarrow tNSH^l$ defined as:

$$\overline{rel}(tnsh, t) = tnsh \mathbin{\overline{\cap}} \texttt{0-mask}(\hat{t}),$$

where $\overline{\cap}$ is the negative intersection of two negative sets, as defined in Section 3.2.6. In addition, $\overline{irrel}(tnsh, t)$ is defined as:

$$\overline{irrel}(tnsh, t) = tnsh \mathbin{\overline{\cap}} \texttt{1-mask}(\hat{t}).$$

Because the negative representation is the complement, it is not only more compact for large positive set-sharing instances, but also, and perhaps more importantly, it enables us to use inverse operations that are more memory and computationally efficient than in the positive representation. However, the negative representation does have its limitations. Certain operations that are straightforward in the positive representation are $\mathcal{NP}$-Hard in the negative representation [41, 46]. A key observation given in [41] is that there is a mapping from Boolean formulas to the negative set-sharing domain such that finding which strings are not represented is equivalent to finding satisfying assignments to the corresponding Boolean formula. This is known to be an $\mathcal{NP}$-Hard problem. As mentioned before, this fact is exploited in [41] for privacy enhancing applications. The mapping is defined as follows.

Let $tnsh = \{11^{**}, 1^{*}1^{*}, {}^{*}11^{*}, {}^{**}11\}$ be the same sharing set as in Example 5.5.1. Its equivalent Boolean formula $\phi \equiv$ not [(x_1 and x_2) or (x_1 and x_3) or (x_2 and x_3) or (x_3 and x_4)] is defined over the set of variables $\{x_1, x_2, x_3, x_4\}$. The formula ϕ is mapped into a negative set-sharing instance where each clause corresponds to a string and each variable in the clause is represented as a 0 if it appears negated, as a 1 if it appears un-negated, and as a * if it does not appear in the clause. By applying DeMorgan's law, we can convert ϕ to an equivalent formula in conjunctive normal form. Then, it is easy to see that a satisfying assignment of the formula such as $\{x_1 = true, x_2 = false, x_3 = false, x_4 = true\}$ corresponding to the string 1001 is not represented in the negative set-sharing instance.

Theorem 9 *A polynomial time algorithm for computing negative cross-union,* $\overline{\otimes}$ *, implies* $\mathcal{P}$=$\mathcal{NP}$.

To show that negative cross-union, $\overline{\otimes}$, is $\mathcal{NP}$-Complete we first restate the definition of Non-Empty Self Recognition ($NESR$) shown to be $\mathcal{NP}$-Complete in [41]. Then, we use $NESR$ to show that there is no polynomial time algorithm for computing negative cross-union unless $\mathcal{P} = \mathcal{NP}$.

(Non-Empty Self Recognition, $NESR$**).**
INPUT: A negative set $tnsh$ of length l strings over the alphabet $\{0, 1, *\}$.
QUESTION: Does $tnsh$ represent an empty positive set bsh? In other words, does there exists a string in $\{0, 1\}^l$ not matched in $tnsh$?

The following is a proof for Theorem 9:

Proof 5.5.1 Given a negative set $tnsh$ of length l, assume a polynomial time algorithm $\mathcal{M}$ that takes as input negative sets $tnsh_1$ and $tnsh_2$ and outputs $tnsh' =$

$tnsh_1 \overline{\boxtimes} tnsh_2$, where $tnsh'$ represents the result of the positive cross-union of the two positive sets represented by $tnsh_1$ and $tnsh_2$.

We construct a polynomial time algorithm for $NESR$: given any instance of $NESR$ with input $tnsh$. First, generate a positive set sh with two strings s_1 and s_2 of length l each having alternating 1's and 0's, e.g., if $l = 4$, then $sh = \{0101, 1010\}$. Convert sh to its negative set representation, nsh, using a polynomial time algorithm, i.e., letting $k = log_2(l)$ or the Prefix algorithm, see [41]. Verify that s_1 and s_2 appear in $tnsh$: if either one is missing from $tnsh$, then answer "No" ($tnsh$ is not empty, at a minimum it represents the missing string). Otherwise, both s_1 and s_2 appear in $tnsh$, but there may be some other string(s) missing from $tnsh$ ($tnsh$ is not empty). Let $\mathcal{M}$ compute $tnsh' = tnsh \ \overline{\boxtimes} \ nsh$. Now, check if both s_1 and s_2 appear in $tnsh'$: if both are missing from $tnsh'$, then answer "Yes" ($tnsh$ is empty); otherwise, answer "No".

Note that if $tnsh$ represented an empty positive set, then its *negative cross-union* with another set nsh will yield a representation of the same set nsh. In other words, if $tnsh$ is empty and since $s1$ and $s2$ were missing from nsh, then $s1$ and $s2$ will also be missing from the result $tnsh'$. On the other hand, if $tnsh$ is *not* empty (represents some string(s), other than $s1$ and $s2$, in the positive), then negative cross-union (ternary bitwise-or operation) with one of the two strings will produce a different string to s_1 or s_2 resulting in either s_1 or s_2 appearing in $tnsh'$. Thus, $\mathcal{M}$ can be used to solve $NESR$ efficiently. Since $NESR$ is $\mathcal{NP}$-Complete, then we have shown $\mathcal{P}$=$\mathcal{NP}$.

Due to the interdependent nature of the relationship between the elements of a negative set, it is unclear how a precise negative cross-union can be accomplished without going through a positive representation. Therefore, we accomplish the negative cross-union by first identifying the represented positive strings and then applying cross-union accordingly.

Rather than iterating through all possible strings in $\mathcal{U}$ and performing cross-union on strings not in $tnsh$, we achieve a more efficient negative cross-union, $\overline{\boxtimes}$, by converting $tnsh$ to tsh first, i.e., using `NegConvert` from Table 5.2 and performing ternary cross-union on strings $t \in tsh$. In this way, the ternary representation continues to provide a compressed representation of the sharing set. Note that the negative up-closure operation, $\bar{*}$, suffers the same drawback as cross-union. Therefore, we handle it in the same way as the negative cross-union.

To further illustrate differences in operating between the positive and negative representation, the following example shows that the result of a ternary cross-union of elements from negative sets is not equivalent to the ternary cross-union of the equivalent positive sets.

Example 5.5.2 Let $tnsh_1 = \{01^*, 10^*\}$, or $\{010, 011, 100, 101\}$ fully specified, and $tnsh_2 = \{010\}$. Their positive ternary cross-union $tnsh = tnsh_1 \boxtimes tnsh_2 = \{010, 011, 110, 111\}$. However, if we take the complement of $tnsh_1$ and $tnsh_2$, $tsh_1 = \{001, 110, 111\}$ and $tsh_2 = \{001, 011, 100, 101, 110, 111\}$, respectively, then $tsh = tsh_1 \boxtimes tsh_2 = \{001, 011, 101, 110, 111\}$. The complement of tsh =$\{010, 100\}$ which is not equivalent to $tnsh = \{010, 011, 110, 111\}$. The correct negative cross-union operation should have returned $tnsh' = tnsh_1 \overline{\boxtimes} tnsh_2 = \{001, 011, 101, 110, 111\}$.

A canonical negative set, in its entirety, represents all the strings not in the positive. For example, let tnsh be a negative set with two negative elements of length 4, $tnsh = *00*, *11*$. Notice that $tnsh$ *contains* all the 4-bit strings with "00" and "11" in the middle two positions, i.e., 0000, 0001, 1000, 1001, 0110, 0111, 1110, 1111. However, it *represents* all 4-bit *positive* strings *except* those with "00" or "11" in the middle two positions, i.e., 0010, 0011, 1010, 1011, 0100, 0101, 1100, 1101. This is $\mathcal{U} \setminus tnsh = bsh$. If we append another negative string, say "1***", all strings with

a "1" in the first position, to the negative set, we obtain $tnsh' = *00*, *11*, 1 * **$. Now $tnsh'$ contains not only the strings with "00" and "11" in the middle two positions (like $tnsh$), but also those with a "1" in the first position, i.e., $tnsh \cup \{1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111\}$. Consequently, $tnsh'$ represents all 4-bit *positive* strings except those with "00" or "11" in the middle two positions AND a "1" in the first position, i.e., 0011, 0100, 0010, 0101. Notice that 1010, 1011, 1100, 1101 were eliminated from the representation. Each non-redundant negative string added reduces the positive strings represented. Thus, the entire negative set is necessary to represent even just one positive element. Due to its relationship with SAT problems [41], we can consider the negative set as equivalent to a Boolean formula that can be solved to reveal the positive strings represented which is, in general, a hard problem.

Due to the conjunctive nature of the relationship between the elements of a negative set, it is unclear how a precise negative cross-union can be accomplished without going through a positive representation. Therefore, we accomplish the negative cross-union by first identifying the represented positive strings and then applying cross-union accordingly.

Bitwise operations are used throughout the negative algorithms to compress, expand, and perform set operations on negative sets while preserving the precise positive sets they represent. As shown by the conversion algorithms, new strings are added to or removed from a negative representation with the `Delete` and `Insert` operations, and these strings are known a priori.

Since cross-union may create new strings as well as eliminate existing ones, *negative cross-union* would need to modify the negative set to both `Delete` existing negative strings to represent any newly created positive strings, as well as `Insert` strings not currently represented to remove them from the positive. Since a single negative string only guarantees that a particular group of subsumed strings are not in the positive, the entire negative set (taken as a whole) is required to repre-

sent the positive strings. Therefore, we can only accomplish negative cross-union by first identifying all the positive strings represented and performing cross-union accordingly.

In general, bitwise operations that may introduce new strings in the positive set cannot be applied directly to the elements in a negative representation to achieve an equivalent result in the positive representation. Since the negative representation contains all elements *not* in the set, a single element in the negative set does not guarantee that a specific positive string is represented. Furthermore, a single negative element only guarantees that this particular string is *not* in the positive. Since ternary-or creates an entirely new string, we cannot determine if the new negative string is actually in the positive result. Therefore, the entire negative set is required to determine which positive strings the bitwise operator should be applied to.

Definition 5.5.2 (Negative abstract unification, $\overline{amgu}$). The *negative abstract unification* is a function $\overline{amgu} : \mathcal{V} \times Term \times tNSH^l \rightarrow tNSH^l$ defined as

$$\overline{amgu}(x, t, tnsh) = \overline{irrel}(tnsh, x = t) \,\overline{\cup}\, (\overline{rel}(tnsh, x) \;\overline{\bowtie}\; \overline{rel}(tnsh, t))^{\overline{*}},$$

where $\overline{\cup}$ is the negative set union as defined in Section 3.2.2.

Example 5.5.3 (**Negative abstract unification**). Let $tnsh$ = {11**, 1*1*, *11*, **11} be the same sharing set as in Example 5.5.1. Consider the analysis of $X_1 = f(X_2, X_3)$:

$$
\begin{array}{lcl}
A = \overline{rel}(tnsh, X_1) & = & \{11**, 1*1*, *11*, **11, 0***\} \\
B = \overline{rel}(tnsh, f(X_2, X_3)) & = & \{11**, 1*1*, *11*, **11, *00*\} \\
A \mathbin{\overline{\boxtimes}} B & = & \{00**, 01**, 0*0*, *00*\} \\
(A \mathbin{\overline{\boxtimes}} B)^{\overline{*}} & = & \{01**, 0*1*, 100*\} \\
C = \overline{irrel}(tnsh, X_1 = f(X_2, X_3)) & = & \{11**, 1*1*, *11*, **11, 1***, \\
 & & \quad *1**, **1*\} \\
 & = & \{1***, *1**, **1*\} \\
\overline{amgu}(X_1, f(X_2, X_3), tnsh) = C \mathbin{\overline{\cup}} (A \mathbin{\overline{\boxtimes}} B)^{\overline{*}} & = & \{01**, 0*1*, 0**0, 100*\}
\end{array}
$$

Definition 5.5.3 (Negative projection, $\overline{tnsh|_t}$). The *negative projection* is a function $\overline{tnsh|_t}$: $tNSH^l \times Term \rightarrow tNSH^k$ $(k \leq l)$ that selects elements of $tnsh$ projected onto the binary representation of $t \in Term$ and is defined as

$$\overline{tnsh|_t} = \overline{\pi}(tnsh, \Upsilon_t),$$

where Υ_t is equal to all i^{th}-bit positions of $\hat{t}$ where $\hat{t}[i] = 1$ and $\overline{\pi}$ is the negative project operation, as defined in Section 3.2.7.

Example 5.5.4 (**Negative projection**). Let $tnsh$ = {11**, 1*1*, *11*, **11} be the same sharing set as in Example 5.5.1. The negative projection of $tnsh$ over the term $t = f(X_1, X_2, X_3)$ is $\overline{tnsh|_t}$ = {11*, 1*1, *11}. String **1 is not in the result because it represents the following strings when fully specified {001, 011, 101, 111} and not all these strings are in the complement, e.g., 001 is in the positive result of the same projection over bsh.

Definition 5.5.4 (Negative initial state, $\overline{init}$). The *negative initial state* $\overline{init}$: $\mathcal{V} \times \mathcal{I}^+ \rightarrow tNSH^{|\mathcal{V}|}$ describes an initial substitution given a set of variables of interest. Assuming as in Definition 5.4.7 the binary initial state operation $init_{bSH}$:

bsh	$tnsh_1$	tsh_1	$tnsh_2$	tsh_2
0011	000*	0100	000*	0100
0100	0*01	1**1	0*01	0*11
0111	1**0	**11	101*	11*1
1001	*0*0		1**0	1*01
1011	**10		*0*0	*111
1101			**10	
1111				

Table 5.3: $tnsh_1$ and $tnsh_2$ *contain* different strings but *represent* the same *bsh*. Also, their respective complements, tsh_1 and tsh_2, differ in the elements they *contain* but *represent* the same *bsh*.

$\mathcal{V} \rightarrow bSH^{|\mathcal{V}|}$, the negative initial state can be defined using either `NegConvert` or `NegConvertMissing` described in Fig. 5.4 and denoted both by $\overline{\texttt{Convert}}$ as follows:

$$\overline{init}(\mathcal{V}, k) = \overline{\texttt{Convert}}(init_{bSH}(\mathcal{V}), k)$$

Definition 5.5.5 (Negative set equivalence, $\equiv$). Given $tnsh_1, tnsh_2 \in tNSH^l$, they are *equivalent* if and only if $(\forall t_1 \in tnsh_1, \forall s_1 \trianglelefteq t_1, s_1 \not\trianglelefteq tnsh_2) \wedge (\forall t_2 \in tnsh_2, \forall s_2 \trianglelefteq t_2, s_2 \not\trianglelefteq tnsh_1)$.

As one can see from the example in Table 5.3, more than one negative set, $tnsh_1$ and $tnsh_2$, can represent the same postive set, *bsh*. This presents further complications in comparing negative sets for equivalency.

Definition 5.5.6 (Negative join, $\overline{\sqcup}$). Given $tnsh_1, tnsh_2 \in tNSH^l$, the *negative join* function $\overline{\sqcup} : tNSH^l \times tNSH^l \rightarrow \wp^0(tNSH^l)$ is defined as the negative set union of the two sets, i.e., $tnsh_1 \,\overline{\cup}\, tnsh_2$.

This section presented an alternative representation to positive set-sharing instances, namely using its complement. Taking advantage of the smaller complement

size further improved the results from the positive ternary representation allowing for completion of abstract unification more efficiently, see Section 5.6. New conversion algorithms were described and associated operators were redefined to accommodate this new "negative" representation. The next section compares the results gathered from multiple experiments using all three sharing set representations.

5.6 Experimental Results

A proof-of-concept implementation was developed in order to measure experimentally the relative efficiency in terms of running time and memory usage obtained with the two new representations described earlier, tSH and $tNSH$. The prototype uses *Patricia tries* [86] to handle efficiently binary and ternary strings, and is connected to a naive *bottom-up* fixpoint analyzer.

Our first objective is to study the implications of the conversions in the representation for analysis. Note that although both tSH and $tNSH$ do not imply a loss of precision, the sizes of the resulting representations and their conversion times can vary significantly from one to another. An essential issue is to determine experimentally the best overall k parameter for the conversion algorithms. Second, we study the core abstract operation of the traditional set-sharing, $amgu$, under two different metrics. One is the running time to perform the abstract unification. The other metric expresses the memory usage through the size of the representation in terms of number of strings during key steps in the unification. All experiments have been conducted on an Intel^R Core^{TM} Duo CPU T2350 at 1.86GHz with 1GB of RAM running Ubuntu 7.04, and were performed with 12-bit strings since we consider this value large enough to show all the relevant features of our approach. In general, within some upper bound, the more variables considered the better the expected efficiency.

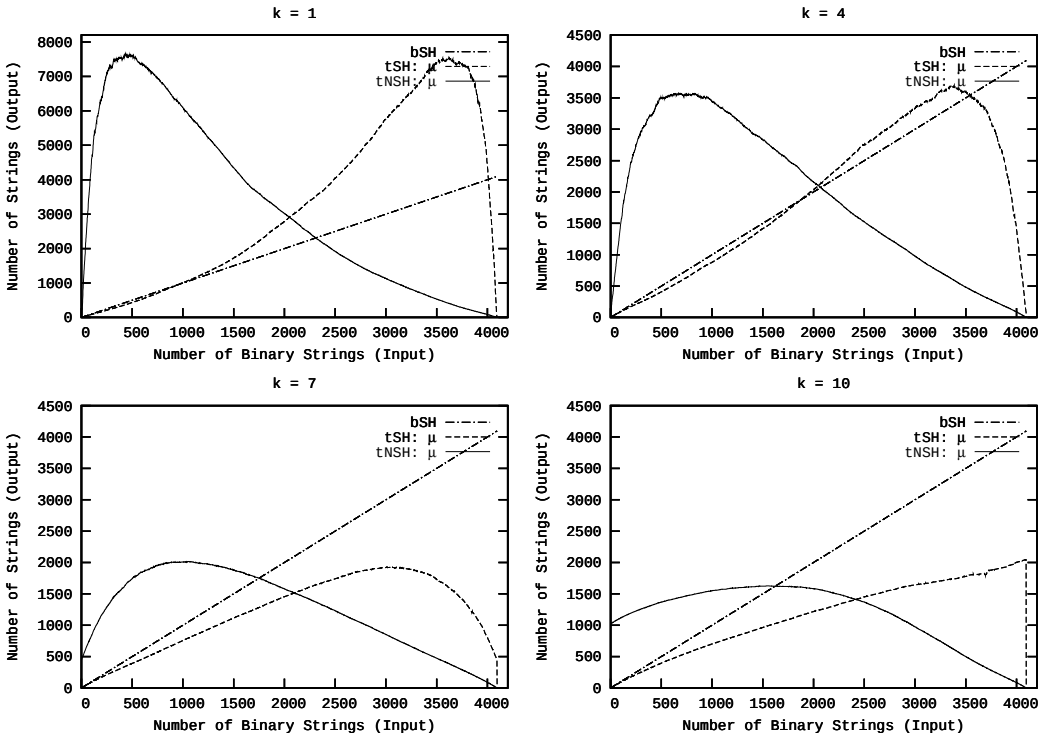

Figure 5.7: Size comparisons, average (μ), for binary (bSH), ternary (tSH), and negative ternary ($tNSH$) with $l = 12$ for k = 1, 4, 7, and 10.

The first experiment determines the best k value suitable for the conversion algorithms, shown in Figs. 5.1 and 5.4. We proceed by submitting a set of 12-bit strings in random order using different k values. We evaluate size for the smallest output (see Fig. 5.7) for a given k value. As expected, bSH ($x = y$ line) results in no compression; tSH slowly increases with increasing input size, remaining below bSH (for $k = 7$ and $k = 10$) due to the compression provided by the $*$ symbol and by having little redundancy; $tNSH$, the complement set, starts larger than bSH but quickly tapers off as the input size increases past 50% of $|\mathcal{U}|$. Since the k parameter helps determine the minimum number of specified bits in the set, there is a direct relationship between the k parameter and the size of the output due to compression by the $*$ symbol. A smaller k value, i.e., $k = 1$, introduces the maximum number of $*$ symbols in the set. However, for a given input, a small k value does not necessarily

result in the best compression factor (see $k = 1$ of Fig. 5.7). This result may be counter-intuitive, but it is due to the potentially larger number of unmatched strings that must be re-inserted back into the set determined by all the strings that must be represented by the converted result, see line 13-17 of Fig. 5.4. In addition, a small k value may result in a set with more ternary strings than the number of binary strings represented. This occurs when multiple ternary strings, none of which subsumes any other, represent the same binary string. This redundancy in the ternary representation is not prevented by `ManagedGrowth`, and is apparent in Fig. 5.7 when $|tSH|$ and $|tNSH|$ exceed the maximum size of binary sharing relationships (i.e., 4096). One way to reduce the number of redundant strings is to sort the binary input by *Hamming distance* before conversion. In the subsequent tests, sorting was performed to maximize compression. We have found empirically that a k setting near (or slightly larger than) $l/2$ is the best overall value considering both the result size and time complexity. We use $k = 7$ in the following experiments. It is interesting to note that a k value of $log_2(l)$ results in polynomial time conversion of the input (see the Complexity column of Table 5.2) but it may not result in the maximum compression of the set (see $k = 4$ of Fig. 5.7). Therefore, k may be adjusted to produce results based on acceptable performance level depending on which parameter is more important to the user, the level of compression (memory constraints) or execution time.

Our second experiment shows the comparison in terms of memory usage (Fig. 5.8, left) and running time (Fig. 5.8, right) of the conversion algorithms for transforming an initial set of binary strings, bSH, into its corresponding set of ternary strings, tSH, or its complement (negative), $tNSH$. We generated random sets of binary strings (over 30 runs) using $k = 7$ and we converted the set of binary strings using the `Convert` algorithm described in Fig. 5.1 for tSH, and `NegConvertMissing` in Fig. 5.4 for $tNSH$. We also reduced the number of redundant strings by sorting them using the Hamming distance before conversion. The plot on the left shows that the number of positive ternary strings,$|tSH|$, used for encoding the input binary strings

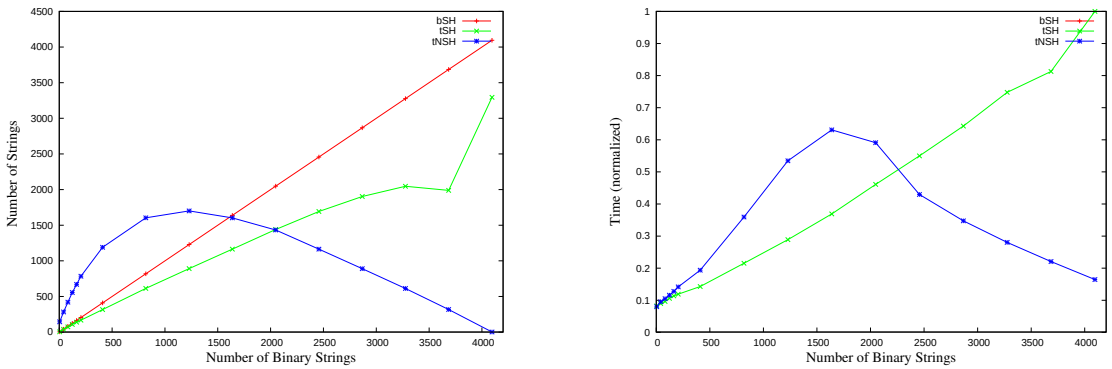

Figure 5.8: Memory usage (avg. # of strings) and time normalized for conversions with $k = 7$.

always remains below $|bSH|$, and this number increases slowly with increasing input size. It important to notice that for large values of $|bSH|$, tSH compacts worse than expected and the compression factor is lower. The main cause is the use of the parameter $k = 7$ that implies only the use of 5 or less $*$ symbols for compression. Conversely, the number of negative sharing relationships, $|tNSH|$, is greater than $|bSH|$ and $|tSH|$ up to between 40% and 50%, respectively. However, when the load exceeds those thresholds $tNSH$ compresses much better than its alternatives. For instance, for the maximum number of binary sharing relationships, $tNSH$ compresses them to only one negative string. On the other hand, the rightmost plot shows the average time consumed over 30 runs for both conversion algorithms. Again, $tNSH$ scales better than the positive ternary solution, tSH, after a threshold established around 50% of the maximum number of binary sharing relationships.

The proof-of-concept implementation is not really optimized, since our objective is to study the *relative* performance between the three representations, and thus times are normalized to the range $[0, 1]$. We argue that the comparisons we submit between representations are fair since the three cases have been implemented with similar efficiency, and useful since the absolute performance of the base representation

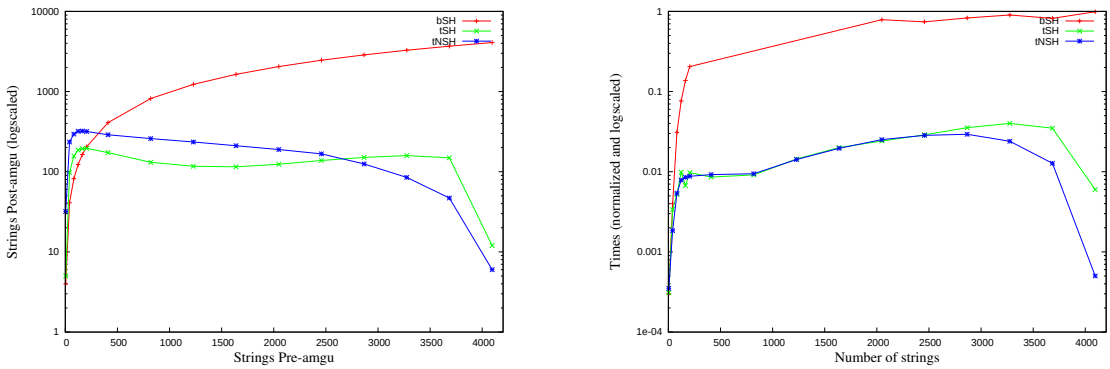

Figure 5.9: Memory usage (avg. # of strings) and time normalized for amgu over 30 runs with $k = 7$.

is well understood.

Finally, our third experiment shows also the efficiency in terms of the memory usage (in Fig. 5.9, left) and running time (in Fig. 5.9, right) when performing the abstract unification for $k = 7$. Several characteristics of the abstract unification influence the memory usage and its performance. Given an arbitrary set of variables of interest $\mathcal{V}$ ($|\mathcal{V}| = 12$), we constructed $x \in \mathcal{V}$ by selecting one variable and $t \in Term$ as a term consisting of a subset of the remaining variables, i.e., $\mathcal{V} \setminus \{x\}$. We tested with different values of t. Another important aspect is the input sharing set, bSH. Again, we reduced the influence of this factor by generating randomly 30 different sets.

In the leftmost plot, the x-axis illustrates the number of input binary strings considered during the *amgu*. In the case of the positive and negative ternary *amgu*, the input binary strings were first converted to their corresponding compressed representations. The y-axis shows the number of strings after the unification. The plot shows that exceeding a threshold lower than 500 in the number of input binary sharing relationships, both tSH and $tNSH$ yield a significant smaller number of strings

than the binary solution after unification. Moreover, when the number of the input binary strings is smaller than 50% of its maximum value, tSH compresses more efficiently than $tNSH$. However, if this value is exceeded then this trend is reversed: the negative encoding yields a better compression as the cardinality of the original set grows toward $2^{|\mathcal{V}|}$.

The rightmost plot shows the size of the random binary input sets in the x-axis, and the average time consumed for performing the abstract unification in its y-axis, normalized again from 0 to 1. This graph shows that the execution times behave similarly to the memory usage during abstract unification. Both tSH and $tNSH$ run much faster than bSH. The differences are significant (a factor of 10) for most x-values, reaching a factor of 1000 for large values of $|bSH|$. When the load exceeds a $50-60\%$-threshold, $tNSH$ scales better than tSH by a factor of 10. The main difference with respect to the memory usage depicted in the leftmost plot is that for a smaller load, tSH runs as fast as $tNSH$ during unification. The main reason is that the ternary relevant and irrelevant sharing operations are less efficient than their negative counterparts: intersection is an expensive operation in the positive ternary representation whereas the negative intersection is very efficient (positive union).

5.7 Conclusions and future work

This chapter presented a novel approach to Set-Sharing that leverages the complement or negative sharing relationships of the original sharing set, without any loss of accuracy. In this work, the negative representation is based on ternary strings. We also showed that the same ternary representation can be used as a positive encoding to efficiently compact the original binary sharing set. This functionality provides the option of working with whichever set sharing representation is more efficient for a given problem instance.

The capabilities of our negative approach to compress sharing relationships are orthogonal to the use of the ternary representation. Hence, the negative relationships may be encoded by any other representation such as, e.g., Binary Decision Diagrams. Concretely, *Zero-suppressed Binary Decision Diagrams* (ZBDDs) [60] are particularly interesting because ZBDDs were designed to represent sets of combinations (i.e., sets of sets). In addition, this approach may be also applicable to similar sharing-related analyses in object-oriented languages, see [81].

Our experimental evaluation has shown that our approach can reduce significantly the memory usage of the sharing relationships and the running time of the abstract operations, including the abstract unification. Our experiments also show how to set up key parameters in our algorithms in order to control the desired compression and time complexities. We have shown that we can obtain a reasonable compression in polynomial time by tuning appropriately those parameters. Thus, we believe our results can contribute to the practical, scalable application of Set-Sharing.

In this research, we go beyond data security applications of negative databases to research ways to reduce the representation of large combinatorial problem instances, e.g., set-sharing analysis. Solving problems with large instances, near the power set of the number of variables, is generally limited by memory and/or execution time. In this application, negative representation has enabled larger set-sharing problem instances to be analyzed. Furthermore, other combinatorial problems amenable to the negative ternary representation may benefit from this research.

Chapter 6

Conclusion

This dissertation explored ways to leverage positive and negative representations of information within the same application. We examined ways to combine both representations to exploit their beneficial properties for use as alternative constructs to ensure data privacy and to solve efficiently larger combinatorial problems.

Chapter 2 introduced the concept of negative databases and provided background information necessary to better understand the materials used in the dissertation. Chapter 3 set the foundation for working with and defining relational operations over negative databases. By defining a negative relational algebra, we improved our understanding of how to manipulate negative databases and can better determine which types of queries are efficient.

Although possessing different properties, a set and its complement contain the same amount of information. Negative databases exploit this characteristic by constantly maintaining a compact representation of the complement set. Queries can be answered by either a positive or negative database by complementing the results accordingly. However, with the negative representation, the positive data represented is not apparent and can be difficult to retrieve. This basic concept enables the privacy

protection of sensitive data as an alternative to cryptographic approaches. In the future, more elegant and efficient relational algorithms and implementation should be explored, along with more advanced database operations.

With the increased dependence on databases, i.e., in electronic commerce, health care management, combined with the costly spread of electronic-based fraud and identity theft, data privacy has become a major concern for companies and individuals alike. Chapter 4 presented an extensible architecture for a negative database management system that can help manage databases storing sensitive data. We take advantage of years of existing research and experience with relational database systems by defining a negative database management system on top of an existing relational database management system. However, we separated functionalities into modules that can be extended or replaced by future enhancements or better implementations. We proposed an architecture that empowers users to selective choose sensitive fields for conversion into hard-to-reverse, singleton negative databases. For efficiency and ease-of-use, non-sensitive data were left in their native positive format.

Today, relational databases are quite common. If given a choice, users will prefer known and easy-to-use systems. I believe that we need a similar system for negative databases to facilitate working with them and to increase its user population. To better utilize negative databases for applications such as private sharing and private matching, we must first be able to understand how to effectively manage negative data in a systematic manner. The $NDBMS$ I propose is a step towards this direction. Future research should explore different internal representations and how they affect the efficiency of storage and query servicing. In addition, other procedural logic implementation should be investigated, e.g., using C, Java, etc., with the goal of implementing a more efficient system.

Chapter 5 presented the bulk of my dissertation research effort using the negative representation of information in a new problem domain. We defined set-sharing

operations over negative data that are precise and more efficient. To this end, we exploited the size of the complement set to represent large set-sharings. We proposed three different representation of set-sharing sets to include binary, ternary and negative ternary sharing. We showed that the negative ternary set-sharing is most efficient of the three representation for large input sets. Future work can improve upon current algorithms and compare our implementations against other Boolean function representations, i.e., reduced-ordered binary decision diagrams.

This dissertation focused on leveraging both positive and negative representations of information to enhance data privacy and to represent large positive sets. In the future, we hope that ideas from negative databases can be applied to more diverse problem areas. For example, sensor networks may benefit from the obfuscated data storage format. Sensor nodes requiring data protection may not use encryption due to resource limitations. In addition, negative database concepts may help in the area of program or circuit obfuscation to protect intellectual property. These applications strive to change the internal program or circuit structures while maintaining the same input/output semantics. Negative databases' ability to morph, while preserving the same the positive set represented may prove useful to this application. Finally, other intractable problems conducive to the negative database representation may benefit from our research findings.

Appendix A

NDBMS Data Model

The following is an alternative way to organize the internal representation of negative data in a relational database management system ($RDBMS$). This was a preliminary study that illustrated a way to represent a negative database in a relational table. Chapter 4 uses a different internal representation selected for its clarity and simplicity.

This internal representation saves storage space since unspecified bit information are not stored. Although showing promise, further research showed that more convoluted query servicing is required to prevent checking unspecified bit values. More specifically, unspecified bits must be resolved during query execution. In addition, negative data with large variances in the number of specified bits cannot be handled efficiently. As a result, in most cases, unspecified bit information was maintained anyway. Therefore, further study is required to take advantage of this or similar representations.

BitOnesTable		BitZerosTable	
BitPos	Tuple	BitPos	Tuple
11	34	5	34
14	34	7	34
21	34	2	35
5	35	7	35
11	35		
21	35		

Table A.1: One way to represent the specified bit values of NDB records. In this example, records 34 and 35 are shown.

A.1 Negative Databases using RDBMS

The following describes one way to incorporate $NDBs$ into $RDBMS$. Initially, several design and implementation approaches were developed to store negative data and to service a basic membership query. This study addressed possible ways to perform a membership query and updates using an $RDBMS$ while storing negative data. It was evident that membership query was going to be frequently used; therefore, it must be efficient. A string is an element of the positive DB only if it does not appear in the NDB, which can mean an exhaustive search operation to find a string match. However, before a query can be serviced efficiently, the negative data and how it is stored must be addressed.

A promising approach is to represent each specified bit in the NDB in two separate tables: one to signify where a tuple has a "1" in a specified bit position and another table to signify where a tuple has a "0" in a specified position. In Table A.1, only the specified bits of records 34 and 35 are maintained. To save storage space only the specified bits (not the $*$'s) are maintained. However, a record identifier is used to specify which record that bit value belongs to. Using specified bit information from both the incoming query string x and NDB, the number of possible matches can be narrowed. Then, to service a membership query, we must either iterate through the

remaining possible NDB candidate tuples and look for a match, or even better, use the full power of the $DBMS$ to perform a structured query language (SQL) query.

This approach starts with a new tuple representation of an NDB bit string. Each record or tuple is given an identifier (an auto-incremented integer). Then, each specified position is kept in a corresponding table. So, as shown in Table A.1, tuples 34 and 35 have a "1" specified in positions 11, 14, 21 and a "0" specified in position 5 and 7. Therefore, tuple 34 is the bit string "****0*0***1**1******1***" and tuple 35 "*0**1*0***1*********1***". By keeping only the specified bit information, a significant amount in storage space is saved, specifically $(l-m)r$ bits, where l is the record length, m is the number of specified bits, and r is the number of records.

A.1.1 Membership Query in an RDBMS

Membership query of a record x is serviced by comparing x with tuples in the NDB—a match means that x is *not* in DB. However, before we can take advantage of SQL, the incoming query string x must also be represented in a similar table format. A procedure transforms the query string into two tables: one for all the position with specified 1's and one for all the specified 0's. We know these tables belong to string x, therefore, we only need to maintain the position numbers (no need for tuple identifiers). Tuple identifiers may be required if we want to use another NDB as the source of query strings, such as in trying to compute the set intersection of two $NDBs$. For example, as shown in Table A.2, string x is the 24-bit string "100111001001111010001011."

It is interesting to note that for each new representation above, the original record can always be recreated. A membership query can be serviced by counting the number of times a specific tuples matched an incoming string x. For a fully specified bit string x (no $*$ symbols), if all 5 specified bits of a certain NDB tuple

OnesTable	ZerosTable
BitPos	BitPos
1	2
4	3
5	7
6	8
9	10
12	11
13	16
14	18
15	19
17	20
21	22
23	
24	

Table A.2: A way to represent specified bit values of an incoming record used in a query. In this example, 24-bit string "100111001001111010001011" is shown.

matched, then string x matches that specific NDB tuple. Since all the $*$'s will match any bit in string x, there is no need to check for a match or even store this information.

An example query used by this approach for the above representation is the following:

```
SELECT ndb.id, ndb.data, ndb.mbits FROM
  (
     -- count number of 1's that matched
     SELECT bot.tuple, count(*) AS cnt
        FROM onestable ot, bitonestable bot
        WHERE ot.position = bot.position
        GROUP BY bot.tuple
     UNION
```

```
        -- count number of 0's that matched
    SELECT bzt.tuple, count(*) AS cnt
        FROM zerostable zt, bitzerostable bzt
        WHERE zt.position = bzt.position
        GROUP BY bzt.tuple
)
AS temp, oldndb AS ndb
    WHERE temp.tuple = ndb.id
    GROUP BY ndb.id
    HAVING sum(cnt) $>=$ ndb.mbits;
```

This seems like a complicated way to perform a membership query, but a prototype implementation returned an answer within a second to query with a 24-bit string against a 175 record negative database. Other matching algorithms may be faster depending on the data representation used to store the negative bit information.

A.1.2 Basic Negative Database Operators

Three basic operations were implemented using MySQL 5.0 and its native procedural language. They were implemented according to the algorithms described in [38]. These operations are essential for establishing and maintaining a negative database. They include Negative Pattern Generate, Insert and Delete.

Negative Pattern Generate (NPG) is the heart of the negative database creation process. It takes a negative database and a string x defined over $\{0, 1, *\}$ and outputs a randomly generated string that matches x. So, no string in the positive DB matches the output of NPG. In addition, the number of specified bits can be specified and stored as metadata. For example, if all the tuples in the NDB have 5 or 6 specified bits, then the resulting output of this procedure is a bit string with 5 or 6 specified

bits that matches x and nothing else in the positive DB. If the number of specified bits in x is already 5 or 6 (or less), then x is returned.

The Insert procedure adds a set of strings into the existing NDB without introducing other unwanted strings. The randomization of selected bit positions plays a major part in ensuring that the resulting set having less discernible patterns are generated for subsequent inserts. If patterns can be discerned, then an adversary may deduce other strings in the NDB and possibly reverse engineer the process to retrieve the positive database. Given a string x with m unspecified bits, the procedures generates 2^r possible strings, $0 < r < m$.

The Delete procedure removes all strings that match x from the given NDB. To ensure that only those strings that match x are removed, other strings may be inserted into the NDB. Specifically, strings that subsume x but also match other strings in NDB must be replaced. These new strings will specify a random number of previously unspecified bits. A important concern is that this procedure may cause the NDB to grow in size (exponential to number of $*$'s).

This preliminary study revealed that it is possible to store a negative data representation in a table format and perform membership queries on negative representations using an $RDBMS$.

References

[1] D. Achlioptas, C. Gomes, H. Kautz, and B. Selman, *Generating satisfiable problem instances*, Proceedings of AAAI-00 and IAAI-00 (Menlo Park, CA), AAAI Press, Jul 2000, pp. 256–261.

[2] C. Aggarwal, *On k-anonymity and the curse of dimensionality*, In proceedings 31st International Conference on Very Large Databases (VLDB'05), 2005.

[3] R. Agrawal, A.Evfimiesvski, and R. Srikant, *Information Sharing Across Private Databases*, Proceedings of PODS, 2003.

[4] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard, *Boolean functions for dependency analysis: Algebraic properties and efficient representation*, Static Analysis Symposium, SAS'94 (Namur, Belgium) (Springer-Verlag, ed.), LNCS, no. 864, September 1994, pp. 266–280.

[5] R. Bagnara, R. Gori, P. M. Hill, and E. Zaffanella, *Finite-tree analysis for constraint logic-based languages*, Information and Computation **193** (2004), no. 2, 84–116.

[6] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella, *Set-sharing is redundant for pair-sharing*, Theoretical Computer Science **277** (2002), no. 1-2, 3–46.

[7] Alberto Belussi, Elisa Bertino, and Barbara Catania, *An Extended Algebra for Constraint Databases*, IEEE Trans. on Knowl. and Data Eng. **10** (1998), no. 5, 686–705.

[8] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, *Public Key Encryption with Keyword Search*, Proceedings of Eurocrypt, 2004.

[9] Boolean Satisfability Research Group at Princeton, *zChaff*, http://ee.princeton.edu/chaff/zchaff.php, 2004.

[10] L. Bordeaux, Y. Hamadi, and L. Zhang, *Propositional satisfiability and constraint programming: A comparative survey*, Tech. report, Microsoft Research (MSR), 2005.

[11] R. Brinkman, L. Feng, J. Doumen, P. H. Hartel, and W. Jonker, *Efficient Tree Ssearch in Encrypted Data*, Information Systems Security Journal **13** (2004), 14–21.

[12] M. Bruynooghe, M. Codish, and A. Mulkers, *Abstract unification for a composite domain deriving sharing and freeness properties of program variables*, Verification and Analysis of Logic Languages (F.S. de Boer and M. Gabbrielli, eds.), 1994, pp. 213–230.

[13] R. E. Bryant, *Graph-based algorithms for Boolean function manipulation*, IEEE Transactions on Computers **C-35** (1986), 677–691.

[14] Randal E. Bryant, *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams*, ACM Computing Surveys **24** (1992), no. 3, 293–318.

[15] R.E. Bryant and Y.-A. Chen, *Verification or Arithmetic Functions with Binary Moment Diagrams*, Tech. Report CMU-CS-94-160, Carnegie Mellon University, Pittsburgh, 1994.

[16] F. Bueno and M. García de la Banda, *Set-Sharing is not always redundant for Pair-Sharing*, 7th International Symposium on Functional and Logic Programming (FLOPS 2004) (Heidelberg, Germany), LNCS, no. 2998, Springer-Verlag, April 2004.

[17] F. Bueno, M. García de la Banda, and M. Hermenegildo, *Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization*, 1994 International Symposium on Logic Programming, 1993.

[18] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, *Private Information Retrieval*, Proceedings of the 36th Annual IEEE Conference on Foundations of Computer Science, 1995.

[19] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, and P. Samarati, *k-anonymity*, Advances in Information Security (2007).

[20] E. F. Codd, *A Relational Model for Large Shared Data Banks*, Communications of the ACM (1970).

[21] _____, *Recent investigations in relational database systems*, Proceedings of the IFIP Congress, 1974.

[22] M. Codish, V. Lagoon, and F. Bueno, *An algebraic approach to sharing analysis of logic programs*, Proc. of the Fourth International Static Analysis Symposium, LNCS, no. 1302, Springer Verlag, 1997, pp. 68–82.

[23] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo, *Improving Abstract Interpretations by Combining Domains*, Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, ACM, June 1993, pp. 194–206.

[24] Michael Codish, Dennis Dams, Gilberto Filé, and Maurice Bruynooghe, *On the design of a correct freeness analysis for logic programs*, The Journal of Logic Programming **28** (1996), no. 3, 181–206.

[25] Michael Codish, Harald Søndergaard, and Peter J. Stuckey, *Sharing and groundness dependencies in logic programs*, ACM Transactions on Programming Languages and Systems **21** (1999), no. 5, 948–976.

[26] S. A. Cook and D. G. Mitchell, *Finding hard instances of the satisfiability problem: A survey*, Satisfiability Problem: Theory and Applications (Du, Gu, and Pardalos, eds.), Dimacs Series in Discrete Mathematics and Theoretical Computer Science, vol. 35, American Mathematical Society, 1997, pp. 1–17.

[27] P. Cousot and R. Cousot, *Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, Fourth ACM Symposium on Principles of Programming Languages, 1977, pp. 238–252.

[28] E. Damiani, S.DeCapitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, *Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs*, ACM CCS 2003, Oct 2003.

[29] G. Danezis, C. Diaz, S. Faust, E. Kasper, C. Troncoso, and B. Preneel, *Efficient Negative Databases from Cryptographic Hash*, In proceedings of the 10th Information Security Conference (ISC'07), 2007.

[30] H. Darwen and C. J. Date, *The third manifesto (databases)*, j-SIGMOD **24** (1995), no. 1, 39–49.

[31] C. J. Date, *A formal definition of the relational model*, SIGMOD Record **13** (1982), no. 1, 18–29.

[32] _____, *An introduction to database systems, 8th edition*, Addison-Wesley, 2003.

[33] C.J. Date, *An Introduction to Database Systems*, Addison-Wesley, Reading, MA, 1995.

[34] G. Davida, D. Wells, and J. Kam, *A Database Encryption System with Subkeys*, ACM Transaction on Database Systems, Jun 1981, pp. 312–328.

[35] L. Nunes de Castro and F.J. Von Zuben, *The Clonal Selection Algorithm with Engineering Applications*, Proceedings of Generic and Evolutionary Computation Conference (Las Vegas, NV), Aug 2000, pp. 36–395.

[36] M. de Mare, *An analysis of certain cryptosystems and related mathematics*, Master's thesis, State University of New York Institute of Technology, 2004.

[37] M. de Mare and R. Wright Secure, *Set Membership using 3SAT*, Proceedings of the Eighth International Conference on Information and Communication Security (ICICS '06), 2006.

[38] F. Esponda, *Negative Representations of Information*, Ph.D. thesis, University of New Mexico, 2005.

[39] ———, *Negative Surveys*, Tech. report, Univ. of New Mexico, http://www.citebase.org/abstract?id=oai:arXiv.org:math/0608176, 2006.

[40] ———, *Hiding a Needle in a Haystack using Negative Databases*, Proceedings of Information Hiding, 2008.

[41] F. Esponda, E. Ackley, S. Forrest, and P. Helman, *On-line Negative Databases (with experimental results)*, International Journal of Unconventional Computing **1** (2005), no. 3, 201–220.

[42] F. Esponda, E. S. Ackley, S.Forrest, and P. Helman, *On-line negative databases*, Proceedings of the 3rd International Conference on Artificial Immune Systems (ICARIS) (Catania, Sicily, Italy) (Giuseppe Nicosia, Vincenzo Cutello, Peter J. Bentley, and Jon Timmis, eds.), Springer-Verlag, Sep 2004, pp. 175–188.

[43] F. Esponda, E.S. Ackley, P. Helman, H. Jia, and S. Forrest, *Protecting Data Privacy through Hard-to-Reverse Negative Databases*, In proceedings of the 9th Information Security Conference (ISC'06) (Springer LNCS, ed.), 2006, pp. 72–84.

[44] F. Esponda, S. Forrest, and P. Helman, *Enhancing Privacy through Negative Representations of Data*, Technical report, University of New Mexico (2004).

[45] ———, *Protecting data privacy through hard-to-reverse negative databases*, International Journal of Information Security (Awaiting publication) (2007).

[46] F. Esponda, E.D. Trias, E.S. Ackley, and S. Forrest, *A relational algebra for negative databases*, Tech. report, University of New Mexico, 2007.

[47] Christian Fecht, *An efficient and precise sharing domain for logic programs.*, PLILP (Herbert Kuchen and S. Doaitse Swierstra, eds.), Lecture Notes in Computer Science, vol. 1140, Springer, 1996, pp. 469–470.

[48] S. Forrest, B. Javornik, R. Smith, and A. Perelson, *Using genetic algorithms to explore pattern recognition in the immune system*, Evolutionary Computation **1** (1993), no. 3, 191–211.

[49] S. Forrest, A. S. Perelson, L. Allen, and R. CheruKuri, *Self-nonself Discrimination in a Computer*, Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy (Los Alamitos, CA), IEEE Computer Society Press, 1994.

[50] M. Freedman, K. Nissim, and B. Pinkas, *Efficient Private Matching and Set Intersection*, Advances in Cryptology – Eurocrypt '2004 Proceedings, LNCS 3027, Springer-Verlag, May 2004, pp. 1–19.

[51] H. Garcia-Molina, J. Ullman, and J. Widom, *Database systems: The complete book*, Prentice-Hall, 2001.

[52] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin, *Protecting Data Privacy in Private Information Retrieval Schemes*, Proceedings of STOC, 1998.

[53] G.Liang and S.Chawathe, *Privacy-Preserving Inter-Database Operations*, Proceedings of 2nd Symposium on Intelligence and Security Informatics, 2004.

[54] E. Goh, *Secure Indexes*, In the Cryptology ePrint Archive, Report 2003/216, 2004, pp. 203–216.

[55] Gopal Gupta, Enrico Pontelli, Khayri A. M. Ali, Mats Carlsson, and Manuel V. Hermenegildo, *Parallel Execution of Prolog Programs: a survey*, Programming Languages and Systems **23** (2001), no. 4, 472–602.

[56] H. Hacgumus, B. Iyer, and S. Mehrotra, *Providing Database as a Service*, In Proc. of ICDE, 2002.

[57] H. Haciqumus, B. Iyer, C. Li, and S. Mehrotra, *Executing SQL over Encrypted Data in the Database-service-provider Model*, Proceedings of ACM SIGMOD 2002, Jun 2002.

[58] P. M. Hill, E. Zaffanella, and R. Bagnara, *A correct, precise and efficient integration of set-sharing, freeness and linearity for the analysis of finite and rational tree languages*, Theory and Practice of Logic Programming **4** (2004), no. 3, 289–323.

[59] J. Horey, M. Groat, S. Forrest, and F. Esponda, *Anonymous Data Collection in Sensor Networks*, The 4th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, 2007.

[60] Shin ichi Minato, *Zero-suppressed bdds for set manipulation in combinatorial problems*, DAC '93: Proceedings of the 30th international conference on Design automation (New York, NY, USA), ACM, 1993, pp. 272–277.

[61] D. Jacobs and A. Langen, *Static Analysis of Logic Programs for Independent And-Parallelism*, Journal of Logic Programming **13** (1992), no. 2 and 3, 291–314.

[62] J. Jaffar and J. Lassez, *Constraint logic programming*, POPL, 1987, pp. 111–119.

[63] C.A. Janeway, P. Travers, and M. Walport, *Immunobiology (5th edition)*, Garland Publishing, New York, 2004.

[64] H. Jia, C. Moore, and D. Strain, *Generating Hard Satisfiable Formulas by Hiding Solutions Deceptively*, AAAI, 2005.

[65] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz, *Constraint query languages*, Symposium on Principles of Database Systems, 1990, pp. 299–313.

[66] M. Karnaugh, *The map method for synthesis of combinational logic circuits*, Trans. AIEE (1953), 593–598.

[67] H. A. Kautz, Y. Ruan, D. Achlioptas, C. Gomes, B. Selman, and M. E. Stickel, *Balance and filtering in structured satisfiable problems*, IJCAI, 2001, pp. 351–358.

[68] A. King and P. Soper, *Depth-k Sharing and Freeness*, International Conference on Logic Programming, MIT Press, June 1994.

[69] L. Kissner and D. Song, *Privacy Preserving Set Operations*, Proceedings of Advances in Cryptography, 2005.

[70] A. Langen, *Advanced techniques for approximating variable aliasing in Logic Programs*, Ph.D. thesis, Computer Science Dept., University of Southern California, 1990.

[71] *Barton-Dingell Act of 2005*, 2005.

[72] *Personal Data Privacy and Security Act of 2005*, 2005.

[73] *Health Insurance Portability and Accountability Act (HIPAA) of 1996*, 1996.

[74] *California Senate Bill 1386*, 2003.

[75] *Patriot Act of 2001*, 2001.

[76] *Sarbanes-Oxley Act of 2002*, 2002.

[77] X. Li, A. King, and L. Lu, *Collapsing Closures*, ICLP'06, Springer-Verlag, 2006, pp. 148–162.

[78] ———, *Lazy Set-Sharing Analysis*, International Symposium on Functional and Logic Programming, Springer-Verlag, 2006.

[79] U. Maurer, *The Role of Cryptography in Database Security*, Proceedings of ACM SIGMOD, Jun 2004.

[80] E.J. McCluskey, *Minimization of boolean functions*, Bell System Technical Journal, (1956), 1417–1444.

[81] M. Méndez-Lojo and M. Hermenegildo, *Precise Set Sharing Analysis for Java-style Programs*, 9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08), LNCS, Springer-Verlag, January 2008.

[82] R. C. Merkle and M. E. Hellman, *Hiding information and signatures in trapdoor knapsacks*, IEEE-IT **IT-24** (1978), 525–530.

[83] A. Meyerson and R. Williams, *On the Complexity of Optimal K-Anonymity*, ACM PODS, 2004.

[84] S. Minato, *Zero-suppressed BDDs and their applications*, STTT **3** (2001), no. 2, 156–170.

[85] D. Mitchell, B. Selman, and H. Levesque, *Problem solving: Hardness and easiness - hard and easy distributions of SAT problems*, Proceeding of the 10th National Conference on Artificial Intelligence (AAAI-92), San Jose, California, AAAI Press, Menlo Park, California, USA, 1992, pp. 459–465.

[86] Donald R. Morrison, *Patricia: Practical algorithm to retrieve information coded in alphanumeric*, J. ACM **15** (1968), no. 4, 514–534.

[87] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and Sh. Malik, *Chaff: Engineering an Efficient SAT Solver*, Proceedings of the 38th Design Automation Conference (DAC'01), June 2001.

[88] A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe, *On the Practicality of Abstract Equation Systems*, International Conference on Logic Programming, MIT Press, June 1995.

[89] K. Muthukumar and M. Hermenegildo, *Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation*, ICLP, 1991.

[90] ———, *Compile-time Derivation of Variable Dependency Using Abstract Interpretation*, Journal of Logic Programming **13** (1992), no. 2/3, 315–347.

[91] Kalyan Muthukumar, *Compile-time algorithms for efficient parallel implementation of logic programs*, Ph.D. thesis, University of Texas at Austin, August 1991.

[92] J. Navas, F. Bueno, and M. Hermenegildo, *Efficient top-down set-sharing analysis using cliques*, Eight International Symposium on Practical Aspects of Declarative Languages, LNCS, no. 2819, Springer-Verlag, January 2006, pp. 183–198.

[93] A. M. Odlyzko, *The rise and fall of knapsack cryptosystems*, Cryptology and Computational Number Theory (Carl Pomerance and S. Goldwasser, eds.), Proceedings of symposia in applied mathematics. AMS short course lecture notes, vol. 42, pub-AMS, 1990, pp. 75–88.

[94] PostgreSQL Development Open-source Community, *PostgreSQL*, http://www.postgresql.org, 2008.

[95] W. V. Quine, *A way to simplify truth functions*, American Mathematical Monthly (1955), 627–631.

[96] Peter Z. Revesz, *Constraint databases: A survey*, Selected Papers from a Workshop on Semantics in Databases (London, UK), Springer-Verlag, 1998, pp. 209–246.

[97] D Richards, *Data compression and gray-code sorting*, Inf. Process. Lett. **22** (1986), no. 4, 201–205.

[98] J. Sahadi, *Over 40m credit cards hacked*, CNN/Money, Jun 2005.

[99] P. Samarati and L. Sweeney, *Generalizing Data to Provide Anonymity when Disclosing Information*, In proceedings ACM PODS, 1998.

[100] B. Schneier, *Applied Cryptography, Second Edition*, Wiley and Sons, Inc., New York, NY, 1996.

[101] Consumer Sentinel and Identity Theft Data Clearinghouse, *Consumer Complaint and Identity Theft Data, 2005*, Tech. report, Federal Trade Commission, Washington, D.C., 2006.

[102] P. Shaw, K. Stergiou, and T. Walsh, *Arc consistency and quasigroup completion*, In Proceedings of ECAI98 Workshop on Non-binary Constraints, 1998.

[103] A. Silberschatz, H. Korth, and S. Sudarshan, *Database System Concepts (5th edition)*, McGraw Hill, 2006.

[104] H. Søndergaard, *An application of abstract interpretation of logic programs: occur check reduction*, European Symposium on Programming, LNCS 123, Springer-Verlag, 1986, pp. 327–338.

[105] D. Song, D. Wagner, and A. Perrig, *Pratical Techniques for Searches on Encrypted Data*, IEEE Symposium on Security and Privacy, 2000, pp. 44–55.

[106] ———, *Search on Encrypted Data*, Proceedings of IEEE SRSP, 2000.

[107] L. Sweeney, *K-Anonymity: A Model for Protecting Privacy*, International Journal of Uncertain Fuzziness Knowledge-based Systems, 2002.

[108] E. Trias, J. Navas, E.S. Ackley, S. Forrest, and M. Hermenegildo, *Negative Ternary Set-sharing*, 24th International Conference on Logic Programming, Dec 2008.

[109] ———, *Two Efficient Representations for Set-sharing Analysis in Logic Programs*, Workshop for Functional and Logic Programming, Jul 2008.

[110] E. Tsang, *Foundations of constraint satisfaction*, Academic Press, 1993.

[111] Staff Writer, *Previous data theft unreported*, http://www.consumeraffairs.com, Apr 2005.

[112] A. Yao, *Protocols for secure computation*, 23rd Annual Symposium on Foundations of Computer Science, November 3–5, 1982, Chicago, IL (1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA) (IEEE, ed.), IEEE Computer Society Press, 1982, pp. 160–164.

[113] E. Zaffanella, R. Bagnara, and P. M. Hill, *Widening Sharing*, Principles and Practice of Declarative Programming (G. Nadathur, ed.), Lecture Notes in Computer Science, vol. 1702, Springer-Verlag, Berlin, 1999, pp. 414–432.

[114] S. Zhong, Z. Yang, and R. Wright, *Privacy-enhancing k-anonymization of customer data*, PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, 2005, pp. 139–147.