A Relational Algebra for Negative Databases

Fernando Esponda
Yale University
P.O. Box. 208285
New Haven, CT 06520-8285
fesponda@cs.yale.edu

Eric D. Trias
University of New Mexico
MSC01 1130
Albuquerque, NM 87131-0001
trias@cs.unm.edu

Elena S. Ackley
University of New Mexico
MSC01 1130
Albuquerque, NM 87131-0001
elenas@cs.unm.edu

Stephanie Forrest
University of New Mexico
MSC01 1130
Albuquerque, NM 87131-0001
forrest@cs.unm.edu

Abstract

A negative database is a representation of all elements not contained in a given database. A negative database can enhance the privacy of sensitive information without resorting to encryption. This can be useful in settings where encryption is too expensive, e.g., some sensor networks, or for applications where searches or other operations on stored data are desired. The original negative database framework supported only authentication queries and operations for modifying data, such as insert and delete. This paper extends that work by defining a set of relational operators for negative representations. For each relational operator, the corresponding negative operator is defined such that the result of the negative operator applied to a negative representation is equivalent to the positive version applied to the positive representation. Algorithms for each relational operator are described and compared to its positive counterpart. This work enhances the practicality of negative databases and expands their range of application.

1. Introduction

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—notably the relational algebra [8] treats databases as sets and defines operators that form the basis for many database management systems. Although databases typically store information of immediate relevance to the task at hand, a different approach stores the complement of the set of interest; that is, 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. Algorithms for creating and storing such sets efficiently are given in [20, 15, 18, 14].

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 access or dynamic databases where records are added and deleted routinely. This privacy-enhancing aspect of negative databases is achieved as a result of the relation between some negative data representations and Boolean satisfiability formulas [18]. The operations defined in this paper allow negative databases to be manipulated without specific knowledge of their contents. For example, in a database containing names and social-security numbers, an operation can be defined to compute (negatively) only the subset of DB that has SSN records from the Southwestern region—the first three digits of American SSNs show the geographical region where the number was issued; two negative databases can be joined by their SSN field to produce a third access list that requires two passwords; or several access lists can be combined into one using only information about the lists' schema.

Applications of negative databases for digital credentials, timestamping, etc., will benefit by the added functionality provided in this paper [14]. Negative databases can be used as proxies for Boolean satisfiability formulas [21], in which case useful operations can be performed on the solutions for the formulas without having to solve the formulas first—this could be of particular interest when the formulas represent constraints to an optimization problem [15].

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 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 using 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 (Section 3.6). Section 5 describes how the complexity of relational operations can be harnessed to support program verification.

In this paper, we provide a stronger foundation for negative databases by bringing them closer to conventional relational databases. Our objective is to increase their practicality and widen the spectrum of potential applications. The original negative database framework supported only authentication queries and operations to modify data, such as insert and delete. The paper extends this earlier work by defining the operations of the relational algebra over negative representations. For each relational operator, the corresponding negative operator is defined such that the result of the negative operator applied to a negative representation is equivalent to the positive version applied to the positive representation. Section 3 gives an algorithm for each operation and its time complexity.

In the remainder of the paper, we first provide some background on negative databases (Section 2). We then describe relational operations on negative databases, giving algorithms and time complexities (Section 3). We then provide an example scenario (Section 4) and describe our prototype implementation, how we are applying it to several problems, some limitations and strategies that address them including empirical results (Section 5), review related work (Section 6), and finally, conclude the paper with a summary of the important points (Section 7).

2. Negative Databases

Negative databases were introduced in [20, 15] as a means to concisely represent all of the binary strings in the complement of a set—referred to as the positive database, denoted as DB—under the assumption of a finite universe, U. The set U of all binary strings of a given length is partitioned in two: positive and negative; the particular choice of partition is a matter of practice, where the positive set collects the strings that have some direct meaning to the application and is usually much smaller than the negative set.

A negative database, NDB, compactly represents the negative image of a set by introducing a special symbol that allows a single NDB entry—a ternary string—to denote many binary strings. This extra symbol is known as the "don't-care" or "wild-card" symbol and written as '*'; its semantics are the usual: a string with a * at position i represents both the string with a 0 and a 1 at position i. A negative database is defined as a set of strings of length n defined over $\{0, 1, *\}$ that *match* all and only the strings in $U - DB$ (see Definition 1 in Section 3), where U is the set of all possible binary strings of length n and DB is a subset of U. Thus, a negative database entry with n * symbols represents 2^n binary strings, and it is said to match each one of them. Table 1 illustrates an example.

DB	$(U - DB)$	NDB
000	001	01*
101	010	11*
	011	0*1
	100	1*0
	110	
	111	

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

Several algorithms for creating negative databases have been proposed, all of which are able to generate an NDB in time polynomial in the size of its input DB [20, 15, 14]. The prefix-algorithm, for example, creates an NDB with at most $n|DB|$ entries in $n|DB|$ time, where $|DB|$ is the number records, each of length n. The different algorithms produce $NDBs$ with different properties depending on number of *'s per string, how difficult it is to recover DB in practice using well-known SAT solvers, and in terms of their size [18]. It was shown in [20] that the general problem of reversing a negative database—recovering the corresponding DB—is $\mathcal{NP}$-hard. As suggested above, however, not all instances are hard in practice (see [18, 14] for examples of how to generate hard instances). More interestingly, easy or hard NDB instances can be constructed depending on application requirements. Finally, note that a negative database might arise in situations where DB is unknown and "negative data" is the only type of data available.

3. 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 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 operator (such as Join or Union) to positive databases. Table 2 gives an example of each relational operator for positive data, and Table 3 gives the corresponding example for negative data. An algorithm and corresponding time complexitym Table 4, are also given for each opera-

tion. Correctness proofs for each algorithm are given in the Appendix.

First, we introduce the following notations and definition:

• x, y, z: strings.
• $x[i, ..., j]$: string x projected onto positions $i, ..., j$.
• U_n: the universe of all possible binary strings of length n.
• Ω_n: ordered list of all n positions for strings of length n.
• Υ_1 and Υ_2: ordered lists of string positions for strings of length n and m, respectively.
• DB_1 and DB_2: subsets of U_n and U_m respectively, referred to as positive databases.
• NDB_1 and NDB_2: negative databases representing $U_n - DB_1$ and $U_m - DB_2$.
• $\sigma_{\Upsilon=v}(DB)$: select operation over a positive database conditioned on v.

Definition 1. Match xMy: Two strings, x and y, match iff $\forall i((x[i] = y[i]) \vee (x[i] = *) \vee (y[i] = *))$

In each case, we first define what a given operation means in terms of the positive and negative sets, i.e., in terms of DB and $U - DB$, respectively. Next, we give a possible semantics using the corresponding negative databases and provide an algorithm that implements the operation and prove its correctness. Lastly, we provide a discussion on the particulars of each operation and its role in an application. We note that the algorithms presented here were chosen for simplicity of exposition rather than for their efficiency.

3.1. Negative Select

The Select operation over a positive database restricts the relation according to some criterion in the form of a predicate. We consider the basic binary operators, $\theta = \{<, \leq, =, \geq, >\}$. 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 limit our description to the latter case, leaving the former for future work. Thus, the Select operation operation applied to DB is defined as:

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

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

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

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

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

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

The above formula shows that Negative Select can be viewed as the Negative Intersection (Section 3.6) 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.

3.1.1 Negative-Equality ($\equiv$) Select

To implement Negative-Equality, it is sufficient to create a negative database that matches every binary string x for which $x[\Upsilon] \neq v$. Theorem 1 in the Appendix proves the correctness of this algorithm:

1. Initialize NDB_2 to the empty set.
2. For each 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 the rest of the positions set to *.
   (b) Append x to NDB_2.

3.1.2 Negative-Less-than ($\overline{<}$) and Negative-Less-than-Equal ($\overline{\leq}$) Select

The following algorithm implements the Negative-Less-than operation. It creates a negative database that matches every binary string x for which $x[\Upsilon] \geq v$.

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 n 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 n for which the positions indicated by Υ have the same value as the corresponding positions in v, and all remaining positions are set to *.
4. Append y to NDB_2.

The proof for this algorithm is established by Theorem 2 in the Appendix. The algorithm to implement the Negative-Less-than-Equal operation can be derived from the algorithm presented above by disregarding step 3. Due to this similarity we forgo its formulation and proof.

3.1.3 Negative-Greater-than ($\overline{>}$) and Negative-Greater-than-Equal ($\overline{\geq}$) Select

The algorithms for these operations are analogous to their $\overline{<}$ and $\overline{\leq}$ counterparts. Below we present the algorithms with the necessary adjustments, although we forgo the proof. The following algorithm creates a database that matches every binary string x such that $x \leq v$:

The algorithms presented below make the necessary adjustments; their proofs, however, are obviated.

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 n 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 n 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 disregarding step 3. Due to this similarity we forgo its formulation and proof.

If we assume that a string can be created in time proportional to its length n and that it takes constant time to copy a string, then the Negative Select operation takes $O(n^2 + |NDB|)$ 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 $O(n^2)$.

To summarize, restricting the contents of a positive database is accomplished by adding elements to its negative image. 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 instance, suppose NDB represents a list of existing customers and their credit card number (CCN) (<name, CCN> tuples). Using NDB prevents sales clerk or order takers from browsing the database to obtain CCNs. If the purchaser is a member of NDB, he should be offered a promotion at checkout (the company has found that offering promotions to first-time customers encourages them to come back). Management may at some point choose to additionally offer promotions to current customers that hold a Visa credit card. This can easily be accomplished, using the operations described above, by restricting NDB to only those records that do not exhibit a Visa credit card, i.e., by appending to NDB records that match every possible <name, CCN> pair for Visa credit card numbers. The prefix of a credit card's number, known as the Bank Identification Number, identifies the credit card network; in this example, a single entry with all positions set to *, except the ones corresponding to the CCN's prefix (set to 4, Visa's prefix) will match all the desired strings. The way in which the promotional database is used does not change—if a <name, CCN> pair is in it, then a promotion is offered. When the promotion for Visa card holders expires, the records can be easily identified and removed.

3.2. Negative Union

We now turn to an operation that is trivially implemented using positive databases, but that requires more care when negative databases are used. For the purpose of this section, we assume that the length of strings in both databases is the same, i.e, $n = m$, and define a new operation over ternary strings:

Definition 2. Coalesce $x \odot y$: Two strings x and y of length n coalesce into string z iff x matches y and for all $1 \leq i \leq n$:

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

The union of two databases can be expressed as:

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

And, the complement is written as:

$$U - DB_1 \cup DB_2 = \{x : x \notin DB_1 \wedge x \notin DB_2\}$$

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

$$NDB_1 \overline{\cup} NDB_2 = \{z : z = x \odot y,\ xMy,\ (x \in NDB_1 \wedge y \in NDB_2)\}$$

The following algorithm produces a new negative database, NDB_3, that realizes $NDB_1 \overline{\cup} NDB_2$. See Table 3 for an example.

1. Initialize NDB_3 to the empty set.
2. For every $x \in NDB_1$:
   (a) For every $y \in NDB_2$ such that x matches y:
      i. Create a string $z = x \odot y$: Let Υ and Υ' be the string positions of x and y respectively that have either a 0 or a 1. Set $z[\Upsilon] = x[\Upsilon]$, $z[\Upsilon'] = y[\Upsilon']$, and, for all $j \in \{\Omega - \Upsilon \cup \Upsilon'\}$, $z[j] = *$.

ii. Append z to NDB_3.

Theorem 3 in the Appendix proves the algorithm's correctness. This operation, is useful in situations where are at least one of the negative databases is hard-to-reverse. Union provides a means to combine negative databases without having to reverse them first, a privacy-enhancing feature.

3.3. Cartesian Product, Join and Intersection

A complete relational algebra as defined in [8] requires only the Cartesian Product; however, we include the Join and Intersection because they are closely related and their implementation, using negative databases, exhibits some interesting subtleties.

Let the generic symbol $\mathtt{op}$ denote one of the following three operators:

1. $\times$: Cartesian Product, when $\Upsilon_2 = \Upsilon_1 = \emptyset$
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 contents of Υ_1 and Υ_2, and by the properties a string has with respect to these positions:

$$U_{n+m-|\Upsilon_2|} = \{xy : \quad x[\Upsilon_1] = z[\Upsilon_2], y = z[\Omega_m - \Upsilon_2], \\ (x \in U_n \wedge z \in U_m)\} \tag{1}$$

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

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

The complement of these operations, referred to as the Negative Cartesian product, Join and Intersection, is:

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

We now discuss how to implement the negative Cartesian product, Join and Intersection, denoted $\overline{\times}, \overline{\bowtie}$ and $\overline{\cap}$ respectively.

The sets U_n, 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 $\{0, 1, *\}$, where the * is the don't care or wild card symbol, and represents the strings in $U_n - DB_1$ in a compact form—a string x with a * in position X_i stands in for strings x' and x'', which are the same as x, except that X'_i is set to 0 and X''_i set to 1; in this way, a string with k * symbols represents 2^k binary strings.

3.4. Negative Cartesian Product

The Negative Cartesian product is defined using regular expressions as:

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

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 has as prefix x and as suffix $m*$ symbols. Append z to NDB_3.
3. For every string $y \in NDB_2$,
   Construct a string z that has as prefix n * symbols and as suffix y. Append z to NDB_3.

An example is shown in Table 2 and Table 3, see Theorem 4 in the appendix for a proof of correctness. This operation 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 doing a positive union of the two.

3.5. 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 2 and 3 for an example).

$$NDB_1 \overline{\bowtie} NDB_2 = \{x*^{m-|\Upsilon_2|} : \ x \in NDB_1\} \\ \cup \{wz : \ w[\Upsilon_1] = y[\Upsilon_2], \\ w[\Omega_n - \Upsilon_1] = *^{|\Omega_n - \Upsilon_1|}, \\ z = y[\Omega_m - \Upsilon_2], y \in NDB_2\} \tag{2}$$

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 $m - |\Upsilon_2|$* as its suffix. Append z to NDB_3.
3. For every string $y \in NDB_2$:
   (a) Create a string $w = *^n$ 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: 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 2 either have their prefix in $U - DB_1$ suffixed with every possible $y = z[\Omega_m - \Upsilon_2], z \in U_m$ (see step 1), or have as suffix $y = z[\Omega_m - \Upsilon_2]$ (step 2(a)) prefixed with every possible string $x = [\Upsilon_1] = z[\Upsilon_2], z \in U - DB_2$ (step 2(b)).

See Theorem 5 in the appendix for a proof of correctness of this algorithm. The Negative Join operation is similar to the Negative Cartesian Product, except that the join condition shrinks each record length accordingly.

3.6. Negative Intersection

The Negative Intersection is defined as:

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

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.

See Theorem 5 in the Appendix for a proof of correctness of this algorithm. An example is shown in Table 2 and 3. Negative Intersection is the simplest of all operators, because it uses de Morgan's Law and computes the union of the two negative databases by appending them.

3.7. Negative Project

The Project operator is a unary operation on a database that returns a subset of the attributes of a relation (here understood as the values at some string positions) as specified and 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 it is the set of attributes not represented in the positive vertical slice—an attribute is in the negative projection if and only if all possible strings of length n with that particular attribute are absent from DB. If we define positive Project as:

$$\pi_\Upsilon(DB) = \{x : \exists_{z \in U_{|\Omega - \Upsilon|}} \exists_{y \in DB} (x = y[\Upsilon] \wedge y[\Omega - \Upsilon] = z)\}$$

Then, its complement is:

$$U_{|\Upsilon|} - \pi_\Upsilon(DB) = \{x : \forall_{z \in U_{|\Omega - \Upsilon|}} \exists_{y \notin DB} \, (\, x = y[\Upsilon] \wedge y[\Omega - \Upsilon] = z \,)\}$$

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

$$\overline{\pi}_\Upsilon(NDB) = \{x : \forall_{w : xMw} \, \forall_{z \in U_{|\Omega - \Upsilon|}} \exists_{y \in NDB} (\, y[\Upsilon] Mw \wedge y[\Omega - \Upsilon] \, Mz \,)\}$$

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 Theorem 7 in the Appendix for the proof. For a special case, a heuristic algorithm, Negative Reduce, has been implemented, see Section 5.

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. This is, NDB has (in general) every possible value for each attribute and we wish to include, as a result of the Negative Project, only the attribute values that are not in any DB record.

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 $\Omega - \Upsilon$—the negative version of selecting all records from DB that have v in $\Omega - \Upsilon$ and then projecting onto Υ. This is accomplished by joining (using the positive equijoin) NDB with a table that has as its single entry v, and then removing all string positions but Υ (projecting onto Υ). The first part of the operation preserves all and only the NDB entries that match strings in $U - DB$ 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 3.1.3.

3.8. Negative Set Difference

The Set Difference operator is a binary operation on two databases, DB_1, 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.

$$DB_1 - DB_2 = \{x | (x \in DB_1) \wedge (x \notin DB_2)\}$$

The complement:

$$U_n - (DB_1 - DB_2) = \{x | (x \notin DB_1) \vee (x \in DB_2)\}$$

Then, Negative Set Difference can be written as:

$$NDB_1 \overline{-} NDB_2 = \{x | (x \in NDB_1) \vee (\forall_{y \in U_n} \neg \exists_{z \in NDB2} (zMy \wedge xMy))\} \quad (3)$$

Similar to Negative Project, there is no polynomial time algorithm that, given as input NDB_1 and NDB_2, outputs

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

Table 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 $\bowtie$ is $\Upsilon_1 = \{2, 3\}$, $\Upsilon_2 = \{1, 2\}$.

NDB_1	NDB_2	$\overline{\sigma}_{\Upsilon=101}(NDB_1)$	$\overline{\times}$	$\overline{\bowtie}$	$\overline{\cap}$	$\overline{\cup}$	$\overline{\pi}_{\Upsilon=1,2}(NDB_2)$	$NDB_1 \overline{-} NDB_2$
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. The results of relational operators on negative databases, NDB_1 and NDB_2. Corresponding results complement those from Table 2. Note: the * symbol stands for both 0 and 1 and the join condition for $\bowtie$ is $\Upsilon_1 = \{2, 3\}$, $\Upsilon_2 = \{1, 2\}$.

a negative database that represents the Negative Set Difference of NDB_1 and NDB_2. Details are shown in Theorem 8 in the Appendix.

As shown above, creating a negative database that represents the Negative Set Difference of two negative databases cannot be realized efficiently in general. However, determining whether a particular string belongs or not to this set can be done with ease. A 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 (see equation 3):

$$x \notin DB_1 - DB_2 \iff \exists_{y \in NDB_1}(yMx) \vee \neg\exists_{z \in NDB_2}(zMx)$$

Conversely, $x \in DB_1 - DB_2 \iff$

$$\neg\exists_{y \in NDB_1}(yMx) \wedge \exists_{z \in NDB_2}(zMx)$$

Applications that hold both negative databases can readily determine the membership of any given string.

The algorithms presented here were chosen for simplicity of exposition rather than for optimality; however, they suffice to illustrate the difference in complexities between a positive and a negative scheme. Table 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 n and that it takes constant time to copy a string.

4. 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 2007. 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 and remain ignorant of participating 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 March 2007: 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

Operation	Select	$\bowtie$ and $\times$	$\cap$	$\cup$
Positive DB	$O(n\|DB\|)$	$O((m+n)(\|DB_1\|\|DB_2\|))$	$O(n\|DB_1\|\|DB_2\|)$	$O(n(\|DB_1\|+\|DB_2\|))$
Negative DB	$O(n^2+n\|NDB\|)$	$O((m+n)(\|NDB_1\|+\|NDB_2\|))$	$O(n(\|NDB_1\|+\|NDB_2\|))$	$O(n\|NDB_1\|\|NDB_2\|)$

Table 4. Comparison of relational operators' asymptotic complexity. Both Negative Project and Negative Set Difference are $\mathcal{NP}$-Hard.

```
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., $NDB_1 \bowtie NDB_2$.[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, no * symbols appear a these positions.

4. 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 of discourse to names (character combinations) with those particular transactions (see Section 5 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. The law enforcement agency can therefore safely eliminate this two fields and send the resulting NDB to CEX.

5. Upon receipt, CEX computes the intersection of its client name 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 NDB_P's contents is unknown and the particular manipulations by the law enforcement agency—restricting transactions to deposits and withdrawals—have been erased.

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

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 separated (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 [17] could also be used to mix the database records and de-identify strings.

5. Implementation

A prototype implementation of each of the relational operator algorithms specified in Section 3, has been developed and tested in an academic environment.

Test data shows that the space complexity of building large negative databases and applying the negative relational operators, is expensive. We are investigating strategies to ameliorate the problem, including distributed negative databases, and the Clean-Up operation introduced in [17]. Here, we introduce some of the motativing applications for each strategy.

One of our motivating applications 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 will be necessary as well. However, mechanisms and algorithms for privacy protection in sensor networks have been lacking. To address these concerns, we developed and evaluated a set of protocols that enable anonymous data collection in a sensor network [24]. 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 actual data. These protocols are collectively referred to as a negative survey [16]. 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 defined in this paper are essential.

A motivating application for the second strategy uses negative databases when the positive dataset approaches the powerset of bit combinations. In this case, it is more efficient to obtain an answer by working with the complement of the problem we intend to solve, and then complementing the solution. For example, this situation arises in some program verification problems [22]. Unlike the sensor network example, this class of problem is concerned less with privacy than with the compact representation of data. This insight opens up the possibility of using easy negative databases to answer questions that might otherwise be intractable.

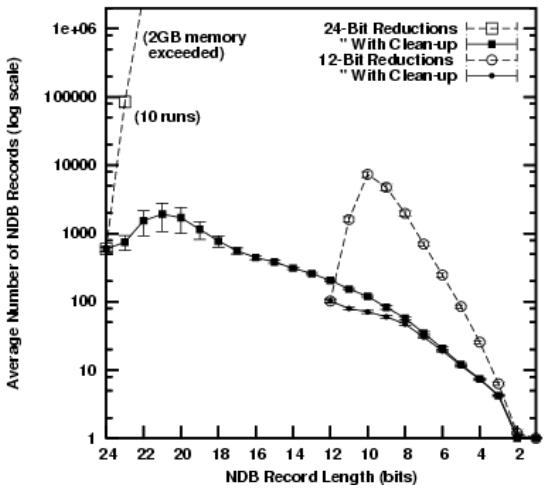

Figure 1. Average NDB size after single bit Negative Reduce on 24- and 12-Bit length records, with and without Clean-Up.

Using an example taken from the program verification problem, we show how the the Clean-Up algorithm 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 vertical 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 followed by a constant number of Clean-Up/Morph operations. 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 1 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 2 shows the average NDB size, alternating between the Negative Reduce and Clean-Up operations, differs as much as three orders of magnitude.

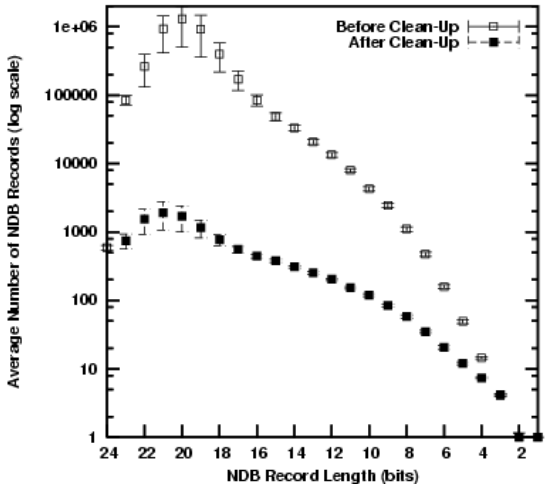

Figure 2. Average NDB size before and after Clean-Up in 24-Bit reduction test.

We suspect that the significant size difference is due to the use of append in the union step instead of the Insert procedure, as discussed in [17], which modifies the entries being inserted into the NDB. 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 1 and 2.

Other applications might require the Select operation to be irreversible and/or for the restriction criteria itself to remain private. Again, we cite the Clean-Up operation 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 [19, 14].

By mitigating the impact of the relational operations on the NDB size with Clean-Up 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, thus enhancing the usability of negative databases. The prototype software is available for download at our website `http://cs.unm.edu/~forrest/negdb.html`.

6. Related Work

Negative databases and relational algebra are the two major sources from which this paper draws. Negative databases are investigated in [20, 15], where it is shown that they can be created efficiently, where a relation with SAT formulas is demonstrated, and its data hiding potential elucidated. The work in [14] 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 [18]. Reference [17] presents a series operations that permit a negative database to be updated, i.e, that allows on-line changes to the contents of DB using only NDB. Relational algebra, on the other hand, is a well-developed research area [8]. Here, we rely only on the operations described in the original references, although further references on the relational algebra include [9, 12, 13, 11].

Other approaches to creating compact representations of binary sets include [27, 34, 29]; also OBDDs (Ordered Binary Decision Diagrams) [6, 7] create a compact representation of binary functions—although there are functions for which the representation size is always exponential—and its many derivatives, such as BMD (Binary Moment Diagrams) [5] and ZBDD (Zero Suppressed Binary Decision Diagram) [31]. Among the differences between these approaches is the need of negative databases to always obtain a compact depiction of the complement of a set without explicitly calculating it, and the ease with which some operations can be performed, e.g., comparing the equivalence of two functions is easy using OBDDs and potentially intractable with negative databases.

Related to this last point is the possibility of using negative databases to enhance privacy in the absence of full cryptography. Other security proposals based on $\mathcal{NP}$-complete problems have been suggested, most notably the Merkle-Hellman cryptosystem [30] based on the general knapsack problem, but most of these schemes have been broken [33]. If a credible level of protection can be achieved using negative databases—and there are ample efforts for creating hard-to-solve SAT instances, e.g., [32, 10, 35, 1, 28, 26]—then, the flexibility of the algebra described in this paper could offset the lack of full cryptographic protection. This could be useful for applications in which limited access to the protected data is required, e.g., for limited searches. Identifying such applications and deepening our understanding of the tradeoff between privacy protection and flexible data access is an area of future investigation.

Our work can also be used to share information across private databases, similar to [2], but without using encryption. Currently, we can efficiently perform membership queries and certain operations (see Table 4) in arbitrarily hard negative database. These can be used to query for keywords in encrypted databases, similar to [36, 3]. It has been shown by [23] that one can perform SQL queries over encrypted databases under a client-server model; more specifically, where the client stores encrypted data at an untrusted server. Since the client owns the data and designed the encryption and exchange protocols, the client is fully aware of how to query its database and decrypt the results. Our model also provides protection from untrusted database servers. Furthermore, it enables any authorized user, even those that did not design the exchange protocols to perform queries against a negative database without having to know decryption keys or mapping functions.

Other avenues for the application of the present work, stemming from the isomorphism NDBs have 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 [25, 37, 4].

7. Summary and Conclusions

The paper describes a closed set of operations that, when applied to negative databases results in an equivalent negative representation as if it was applied to the positive database. We described how the operations of Select, Union, Cartesian Product, Join, Intersection, Project, and Set Difference can be implemented for negative databases; we presented algorithms for the first five operations and proved that no general efficient algorithm exists for the latter two but that implementations are feasible for special interesting cases.

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 data set. The operations discussed here increase the versatility of negative databases by allowing the protected data set 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 an relational algebra defined over them. In particular, we explored 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.

Negative databases will also be useful in situations where no positive data are available or as a proxies for Boolean formulas, in which case the procedures presented here map to manipulations of the formulas themselves, and, more importantly, to implicit manipulations of their solutions—of potential interest given that many problems can be stated in terms of logical formulas.

There are several interesting avenues for future work. They include the design of suitable data structures for negative databases; the optimization of current algorithms and their software implementation; and extending the suite of operations beyond those presented here.

There are several interesting avenues for future work. They include the design of schemata and other data structures for relational negative databases; the optimization of the current relational algebra algorithms and their software implementation; and extending the suite of operations beyond those presented here.

References

[1] D. Achlioptas, C. Gomes, H. Kautz, and B. Selman. Generating satisfiable problem instances. In *Proceedings of AAAI-00 and IAAI-00*, pages 256–261, Menlo Park, CA, July 30–3 2000. AAAI Press.

[2] R. Agrawal, A.Evfimiesvski, and R. Srikant. Information sharing across private databases. In *Proceedings of ACM SIGMOD*, 2003.

[3] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proceedings of Eurocrypt*, 2004.

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

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

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

[7] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[8] E. F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[9] E. F. Codd. Recent investigations in relational database systems. In *Proceedings of the IFIP Congress*, 1974.

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

[11] H. Darwen and C. J. Date. The third manifesto (databases). *j-SIGMOD*, 24(1):39–49, Mar. 1995.

[12] C. J. Date. A formal definition of the relational model. *SIGMOD Record*, 13(1):18–29, 1982.

[13] C. J. Date. *An Introduction to Database Systems, 8th Edition*. Addison-Wesley, 2003.

[14] M. de Mare and R. W. Secure. Set membership using 3sat. In *Proceedings of the Eighth International Conference on Information and Communication Security (ICICS '06)*, 2006.

[15] F. Esponda. *Negative Representations of Information*. PhD thesis, University of New Mexico, 2005.

[16] F. Esponda. Negative surveys. Technical report, http://www.citebase.org/abstract?id=oai:arXiv.org:math/0608176, 2006.

[17] F. Esponda, E. S. Ackley, S. Forrest, and P. Helman. On-line negative databases. In G. Nicosia, V. Cutello, P. J. Bentley, and J. Timmis, editors, *Proceedings of ICARIS*, pages 175–188, Catania, Sicily, Italy, Sep 2004. Springer-Verlag.

[18] F. Esponda, E. S. Ackley, P. Helman, H. Jia, and S. Forrest. Protecting data privacy through hard-to-reverse negative databases. In S. LNCS, editor, *In proceedings of the 9th Information Security Conference (ISC'06)*, pages 72–84, 2006.

[19] F. Esponda, E. S. Ackley, P. Helman, H. Jia, and S. Forrest. Protecting data privacy through hard-to-reverse negative databases. *International Journal of Information Security*, 2007.

[20] F. Esponda, S. Forrest, and P. Helman. Enhancing privacy through negative representations of data. *Technical report, University of New Mexico*, 2004.

[21] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W.H. Freeman & Company, San Francisco, 1978.

[22] G. Gupta, E. Pontelli, K. A. M. Ali, and M. C. Manuel V. Hermenegildo. Parallel execution of prolog programs: a survey. *Programming Languages and Systems*, 23(4):472–602, 2001.

[23] H. Hacigumus, C. L. B. Iyer, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of ACM SIGMOD*, 2002.

[24] J. Horey, M. Groat, S. Forrest, and F. Esponda. Anonymous data collection in sensor networks. In *The 4th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2007.

[25] J. Jaffar and J. Lassez. Constraint logic programming. In *POPL*, pages 111–119, 1987.

[26] H. Jia, C. Moore, and D. Strain. Generating hard satisfiable formulas by hiding solutions deceptively. In *AAAI*, 2005.

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

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

[29] E. McCluskey. Minimization of boolean functions. *Bell System Technical Journal,*, pages 1417–1444, 1956.

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

[31] S. Minato. Zero-suppressed BDDs and their applications. *STTT*, 3(2):156–170, 2001.

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

[33] A. M. Odlyzko. The rise and fall of knapsack cryptosystems. In C. Pomerance and S. Goldwasser, editors, *Cryptology and Computational Number Theory*, volume 42 of *Proceedings of symposia in applied mathematics. AMS short course lecture notes*, pages 75–88. pub-AMS, 1990.

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

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

[36] D. Song, D. Wagner, and A. Perrig. Search on encrypted data. In *Proceedings of IEEE SRSP*, 2000.

[37] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

Appendix

THEOREM 1. *Negative Equality Select ($\equiv$): A binary string x is matched in $NDB_2 \iff x[\Upsilon] \neq v$.*

PROOF.

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 match 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.

□

THEOREM 2. *A binary string x is matched in NDB_2 $\iff x[\Upsilon] \geq v$.*

PROOF.

1. Let x be a binary string matched in NDB_2, and let y be the entry that matches it. 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 that matches $x[\Upsilon]$ 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.

□

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

PROOF.

1. Let x be a string in $U - (DB_1 \cup DB_2)$, $x \in U - DB_1$ and $x \in U - DB_2$. By the definitions of NDB_1 and NDB_2, there is a string $x' \in NDB_1$, and a string $x'' \in NDB_2$ that match x; by transitivity, x' and x'' also match each other, i.e., $x' M x''$. Step i of the algorithm creates a string z such that $z = x' \odot x''$.

   Given that zMx' and $x'Mx$, it follows that zMx. Therefore there is a string in $NDB_1 \bar{\cup} NDB_2$ that matches x.

2. Let x be a binary string matched by some entry z in $NDB_1 \bar{\cup} NDB_2$. By construction, there are strings x' and x'' in NDB_1 and NDB_2, respectively, that match

z and that, by transitivity, also match x. By the definitions of NDB_1 and NDB_2: $x \notin DB_1$ and $x \notin DB_2$. Therefore $x \in U - (DB_1 \cup DB_2)$.

□

THEOREM 4. *Negative Cartesian Product ($\overline{\times}$): A binary string* $x \in U_{n+m} - DB_1 \times DB_2 \iff \exists x'((x' \in NDB_1 \overline{\times} NDB_2) \wedge (x'Mx))$.

PROOF.
Let xy be a binary string in $U_{n+m} - DB_1 \times DB_2$, with $|x| = n$ and $|y| = m$

1. Either $x \in U_n - DB_1$ or $y \in U_m - DB_2$. If $x \in U_n - DB_1$ 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, $x'*^m$, that will match xy. Conversely, if $y \in U_m - DB_2$, there is a string $y' \in NDB_2$ that matches it, and step 3 will generate $*^n y$ that matches xy. Therefore, xy is matched by some entry in $NDB_1 \overline{\times} NDB_2$.

Let xy be a binary string matched by some entry $x'y'$ in $NDB_1 \overline{\times} NDB_2$, with $|x| = |x'| = n$ 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 $xy \in U_{n+m} - DB_1 \times DB_2$.

□

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

PROOF.
Let wz be a binary string in $U_{n+m-|\Upsilon_2|} - DB_1 \bowtie DB_2$, with $|w| = n$ 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' might or might not 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 might or might not 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 $U_{n+m-|\Upsilon_2|}$, (see eq. 1)), that matches w. Step 3(c) creates string xy that matches wz. Therefore wz is matched by some string in $NDB_1 \overline{\bowtie} NDB_2$.

Let wz be a binary string and xy an entry in $NDB_1 \overline{\bowtie} NDB_2$ that matches wz, where $|w| = |x| = n$ 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 $wz \in U_{n+m-|\Upsilon_2|} - DB_1 \bowtie DB_2$.

2. If xy was generated in step 3 then: w is matched by x which is matched by $z'[\Upsilon_2]$ (step 3(a)), for $z' \in NDB_2$; z is matched by y which is matched by $z'[\Omega_m - \Upsilon_2]$ (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 $wz \in U_{n+m-|\Upsilon_2|} - DB_1 \bowtie DB_2$.

□

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

PROOF.

1. Let w be a string in $U_{n+m-|\Upsilon_2|} - DB_1 \cap DB_2$, 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 $w \in U_{n+m-|\Upsilon_2|} - DB_1 \cap DB_2$.

□

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

We will proceed by constructing a polynomial time algorithm for the following $\mathcal{NP}$-complete problem.

Definition 3. Non-empty Self Recognition (NESR):
INPUT: A set NDB of length n strings over the alphabet $\{0, 1, *\}$.
QUESTION: Is DB nonempty? That is, is there some string in $U = \{0, 1\}^n$ not matched by NDB?

NESR was first introduced in [20] and shown there to be $\mathcal{NP}$-complete.

PROOF. Assume there is a polynomial time algorithm $\mathcal{M}$ that takes as input a negative database NDB and a bit position indicator Υ and outputs $\overline{\pi}_{\Upsilon}(NDB)$.
We construct a polynomial time algorithm for NESR: 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 (strings one bit long) answer "No", otherwise answer "Yes". Therefore, since NESR is $\mathcal{NP}$-complete, $\mathcal{P}$=$\mathcal{NP}$. □

THEOREM 8. *Negative Set Difference ($\overline{-}$): 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).

PROOF.
Assume a polynomial time algorithm $\mathcal{M}$ that takes as input two negative databases NDB_1 and NDB_2 and outputs $NDB_1 \overline{-} NDB_2$.
We construct a polynomial time algorithm for NESR: given any instance of NESR with input NDB, let $\mathcal{M}$ compute $NDB' = \emptyset \overline{-} NDB$. 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. Since NESR is $\mathcal{NP}$-complete, $\mathcal{P}$=$\mathcal{NP}$. □