NoDD Algorithm Implementation

Andrew Plum & Nathan Nguyen
Department of Computer Science, University of Idaho
plum0598@vandals.uidaho.edu
nguy7866@vandals.uidaho.edu

KEYWORDS

Normalization and Database Design Theory, Functional Dependency, Normalization, Evaluation Algorithm, and Project 360

ACM Reference Format:
Andrew Plum & Nathan Nguyen. 2023. NoDD Algorithm Implementation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

We were tasked with the creating of an evaluation system with the ability to grade student submitted proofs involving functional dependencies. This project is the next step in the creation of a tutoring system concerning concepts in normalization and database design theory. The majority of our project was developed on the backend. Our work is to be integrated with the existing work of previous projects to reach a complete tutoring and evaluation system. The grading system we created utilizes set definitions of functional dependencies. To understand our methodology, it is required to delve into the theory concerning database design and normalization.

2 BACKGROUND AND THEORY

2.1 Functional Dependencies

In relational database theory, functional dependencies serve as the basis for defining relationships between attributes within a database. A relation in a relational database consists of a set of attributes and a table of tuple-rows i.e. a set of tuples (or rows) where each tuple contains an attribute-value pair for all attributes in the relation. We say two sets of attributes X, Y are functionally dependant $X \longrightarrow Y$ if for every instance of the same X attribute values that occurs within the table, the value of the Y attributes are not distinct. In other words the values of the attributes in X determine the values of the attributes in Y, and Y depends on X. A relation scheme is defined to be a relation paired with a set of functional dependencies. Every relational database is characterized by its underlying relation scheme.

Table 1: Math 101 Students

Student ID	First Name	Last Name	Age
33	Alice	Walnut	19
34	Bob	Cherry	19
35	Alice	Pine	20
36	Charlie	Maple	19

As an example, Table 1: *Math 101 Students* illustrates a basic functional dependency where "Student ID $\longrightarrow$ First Name, Last Name, Age" holds. Note that "Last Name $\longrightarrow$ Student ID" also happens to hold, however if more students were added to the table, this dependency may not continue to be enforced, while the Student ID most likely will continue to discriminate students and determine the other attributes. Also note as a counter example, that "First Name $\nrightarrow$ Age" since the name "Alice" corresponds to two distinct age values, 19 and 20. The relation scheme for this table might rightly be given as
$\langle R(\{`StudentID`, `FirstName`, `LastName`"Age`\}),$
$F = \{`StudentID \longrightarrow \{`FirstName`, `LastName`, `Age`\}\}\rangle$
where R is the relation and F is the set of functional dependencies.

The functional dependencies within a relation determine a unique structure that allows for the elimination of redundancies and the efficient organized large scale storage of data. The theories of database normalization require an understanding of how functional dependencies can be utilized to derive further dependencies and identify minimal super-keys within a relation, allowing for the creation of lossless decomposition of tables and eliminating redundancies while maintaining the overall structure of information.

2.1.1 *Armstrong's Axioms.*

Alongside the formal definitions for functional dependencies on a relation, Armstrong's Axioms or inference rules, provide a logical framework for finding all the functional dependencies of a relational database. There are three primary rules

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

or axioms, and several secondary rules which can be derived from those axioms. Given a relation R with a set of attributes U, and $W, X, Y, Z \subseteq U$, Armstrong's axioms are as follows, beginning with the three primary axioms. (Note that AB is taken to mean $A \cup B$.)

(1) Reflexivity: If $Y \subseteq X$, then $X \rightarrow Y$ holds.
(2) Augmentation: If $X \rightarrow Y$ and $Z \subseteq W$, then $XW \rightarrow YZ$.
(3) Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.
(4) Union Rule: If $X \rightarrow Y$ and $X \rightarrow Z$ hold, then $X \rightarrow YZ$ holds.
(5) Decomposition Rule: If $X \rightarrow YZ$ holds, then $X \rightarrow Y$ and $X \rightarrow Z$ hold.
(6) Pseudo-transitivity Rule: If $X \rightarrow Y$ and $YZ \rightarrow W$ hold, then $XZ \rightarrow W$ holds.

2.1.2 *Proofs Involving Functional Dependencies.*

Various proofs involving functional dependencies can be done all with the goal of normalizing the database. In the tutoring and evaluation system being designed, students will submit these aforementioned proofs so that their understanding of concepts in normalization and database design can be evaluated and guided towards improvement when necessary. The proofs we were tasked with grading were proofs for the logical consequence of F and attribute closure.

2.1.3 *Proofs for the Logical Consequences.*

The logical consequence of F, where F denotes a set of functional dependencies, encompasses all of the functional dependencies which can logically be derived from F. An example of a logical consequence of F would be where F is the set A→B, B→C and where we say A→C is logical consequence of F by transitivity between the first and second given functional dependencies in F. To see if a functional dependency is a logical consequence of F, a proof can be performed where each step in the proof references a rule to demonstrate that the new functional dependency is a logical consequence of F by that rule. If F is a set of functional dependencies, the logical consequence of F involves finding all of the functional dependencies that logically follow from F, F*, which means that the attribute closure of F and F* should be the same.

2.1.4 *Proofs for Attribute Closures.*

An attribute closure is all the possible attributes which can be derived, X^+, when given an initial set of attributes, X, and set of functional dependencies, F. X^+ is found by repeatedly applying set of attributes you currently have to the set of functional dependencies until no more attributes can be added to the current attribute set. Relating this to the previous example, if given $X = A$ and $F = \{A \rightarrow B, B \rightarrow C\}$, the attribute closure X^+ would be $\{A, B, C\}$ as it encompasses all the attributes implicated by the given functional dependencies. A proof can be performed to derive X^+ where each step references what new attributes can be derived while referencing the functional dependency in set F in which it was derived.

3 METHODOLOGY

3.1 Programming Languages and Tools

Since our project was mainly focused on backend operations as well as storing and retrieving data from the database we designed, we decided to use the programming languages PHP and SQL. We also utilized XAMPP, PhpMyAdmin, and GitHub in the development of this project.

3.1.1 *Notepad++ and Visual Studio Code.*

Notepad++ had the advantage of being a lightweight text editor; it offers syntax highlighting and various plugins for coding to streamline the development process. Visual Studio Code on the other hand has even more robust extensions and features to offer than Notepad++ and was used for as an environment for debugging as well as coding. These tools in combination facilitated the coding development process due to their different strengths.

3.1.2 *Git and GitHub.*

Because this was a collaborative team project, we needed some form of version control for the code we developed. We chose to leverage Git through GitHub. GitHub has the benefit of having an easy user-friendly graphical interface to utilize Git. GitHub allowed us to track and manage the changes we made to the code as well as reverting changes to previous versions if needed. For collaborative coding, the pull request and branching features were invaluable for our project as it allowed us to work on separate features simultaneously and merge the contributions seamlessly. It also encourages the development of more modular programming, which is always a desired feature in one's code. These features all ensured a more organized and cohesive codebase. We also utilized GitHub's feature to host our code in a repository there.

3.1.3 *XAMPP.*

XAMPP, a cross-platform web server solution/full-stack, was used for testing and debugging of PHP applications in a local development environment. Because it consists of Apache, MySQL, PHP, and Perl, XAMPP simplified the setup of a complete web server stack on our local machines, and was chosen for its straightforward installation to facilitate a consistent development experience and compatibility between team members and the products.

3.1.4 *PHP.*

Hypertext Preprocessor (also known as PHP for short) is a server-side scripting language that is commonly used for web development. It allows for dynamic content generation and clean integration with web applications on the frontend and databases on the backend, which was a required feature for the

programming language we needed for this project. We could have developed the backend of this project in python, but ultimately chose PHP because most of the preexisting tutoring projects already used PHP so we decided to be consistent with this design choice; it also provided the opportunity to familiarize ourselves with a programming language that we had never used before.

3.1.5 SQL.

For database management and manipulation, we used XAMPP's implementation of MariaDB, a fork of MySQL. This allowed us to execute SQL queries within the project to fetch data. SQL allows for efficient retrieval, insertion, modification, and deletion of data in a database. The language's standardized and natural English like syntax in conjunction with its relational database capabilities all helped in the successful development of the database for the project.

3.1.6 PhpMyAdmin.

PhpMyAdmin was utilized because it offers a graphical user interface for MySQL within the XAMPP environment, making database administration more straightforward and efficient. Its simplified features help us in performing tasks such as executing SQL queries, creating tables, and managing relationships between and within tables. PhpMyAdmin enhanced our ability to understand the underlying data structure of the database because of its ability to visualize and manipulate the database schema.

3.2 NoDD Additions: An Evaluation System

Since the primary objective was to assist in the creation of an automated grading algorithm for a pre-existing "assignment submission and tutoring" system, we elected to begin with a programmatic true/false evaluation of a properly formatted proof. Of the many classes of questions that the NoDD tutoring system will be designed to teach, the focus of our project became centered upon two types: Functional dependency derivation (logical consequence) proofs, and Attribute closure computation proofs.

A logical consequence question tasks students with providing a step-by-step proof via armstrong's axioms that a given set of dependencies implies a given derived dependancy. For the logical consequence proofs, our evaluation strategy was as follows: To determine if the student submitted proof was correct, we first check to see if each of the constituent steps of the proof are valid. Afterwards, we check if the conclusion of the proof reached the desired statement that was to be proven.

We begin by fetching every line of the submitted proof; one line/step contains the logical statement, the inference rule used, and the previous lines referenced. Based on this information we can evaluate whether the step is correct given that the previous steps are correct. Therefore, we used a loop to evaluate the individual steps in sequence, continuing if the statement was logically correct. If all the steps in the proof were determined to be valid, then we would check if the last step of the derivation matched the dependency that was to be derived. If a submission passes all of the step-wise checks as well as the final check, it is determined as being correct and appropriate feedback is given. If it fails any of the checks, the grading program halts at that step and marks if the failure was due to a logical error, or due to a syntactical error.

In order to implement the logical step checks required for these evaluations, we created seven inference rule checking functions, with "given" as a rule zero, and Armstrong's Axioms for the remaining six. The details for how these checks are outlined in section 3.3.2.

For the attribute closure question type, a student is given a set of functional dependencies and tasked with calculating an attribute closure for the given set of attributes. The approach to grading this problem is similar to the approach for the logical consequences problem, step-wise checks followed by a final check, but the details of such checks are different, and the final check operates by using the question information to actually solve the problem, and compare results with the student, rather than using a provided goal to check against. The step-wise checks loop over each step in a student response and sequentially compare the set from the previous step with functional dependency referenced by the student or by set given in the problem statement in the case of the first step; if the functional dependency is implied by the previous step's set, and the newly proposed set is the union of the previous set and the result of the functional dependency, then the check is accepted. For calculating the final answer, the algorithm to calculate the attribute closure is as follows: Beginning with the given set, record the working closure set and loop over each of the given dependencies, if they are implied by the set, update that closure set to include the consequences of the dependency. Repeatedly loop over the given dependencies until the attributes in the closure are the same before and after looping over the dependencies, then halt and return the closure set.

3.2.1 Specific Grading Cases.

Steps that give what could be a correct functional dependency but have an incorrect justification are marked as incorrect. Also, because unnecessary steps (steps which are not referenced by future steps) under the current grading system are marked as valid, they will still lead to the proof being graded as correct even though the proof is not concise. We decided to make this the case because unconcise proofs are still considered just as correct of a proof as a concise proof. One other thing to note is we did not have to worry about marking steps that would be valid if preceded by the correct steps as incorrect if out of order and not preceded by the correct steps

because there would be no way the student could reference the previous steps if they did not already write out those previous steps to be referenced; this was enforced by the nature of the interface the student is working with when writing their proof. If the student did attempt to write a correct proof by writing what is a correct future step, then the step would be marked wrong because it has an incorrect justification because it could not be referencing the correct preceding steps that the student hasn't yet written.

3.3 A Binary Implementation

In order to begin to apply logical checks to the many functional dependencies and sets which we were working with, we decided early on that a string representation would not be ideal for logical comparisons. A uniform approach to all sets within the database would be needed if the logical checks implemented were to be sufficiently modular within the project code. We made the observation that for any subset of some fixed set of attributes, say for example 12 attributes, could be represented as a binary number with that number of bits. As an example if the attribute set is $\{A, B, C, D, E, F, G, H, I, J, K, L\}$ then we can assign each attribute a bit and any subset of this is a binary number. After establishing an ordering convention, if the first attribute listed is the least significant bit, and the second attribute the second least, ect. then we may write $\{A, D, E, F\}$ as 111001 in binary or 57 in decimal. Storing subsets as integers is naturally extendable to sets of cardinality 31 or less, without worries over data-types, and going beyond this to 63 or more is fairly simple, but 31 is more than enough for the needs of the project. A functional dependency is thus reduced to a pair of integers. $ABC \rightarrow DE$ may be represented as left = 7, right = 24; so we have two columns in a database table. The binary representation has an advantage in that it need not take into account attribute names, which are not useful for internal computation, the names may simply be converted to the appropriate bit or vice versa via some lookup table. Another advantage of binary representations is the ability to perform natural low-level mathematical operations on these set representations.

3.3.1 *Binary Operators.*

What was particularly useful about the binary representation of the functional dependencies is that we could use the binary operators when implementing set operations. Here is a list of simple set operations using binary:

(1) Set intersection, $ABC \cap BD = B$, is the same as logical AND, 0111 & 1010 = 0010.
(2) Set union, is the same as logical OR, similar to above.
(3) Equality is preserved. Two integers are equal if and only if their corresponding strings are equal (after appropriate formatting).
(4) Subset comparison, $(AB \subseteq ABC)$ is the same as intersection and equality, (0011 == 0011 & 0111).
(5) Set difference $ABD \setminus ABC = D$ is the same as an XOR and an AND $(1011 \oplus 0111)$ & 1011 = 1000.
(6) Other operations can be easily crafted from combinations above ex: Superset, strict sub/super set, complement, etc.

Through binary operations and boolean logic, we were able to construct functions that can check if a given proof step accurately follows the inference rules that it claims to.

3.3.2 *Operations for Logical Derivation Checks.*

Beginning with the logical checks for inference rules, we designed several complex boolean functions, one for each of Armstrong's axioms. Note that the operation symbols and pseudo-code used are the binary operation symbols used by the C++ programming language unless otherwise clarified. PHP has mostly identical symbol usage for these operations. All of these return true or false, and the variable input names are as follows: "fd" is the newly proposed functional dependency that is being checked, "ref" or "ref1" is the first reference dependency which the new dependency is derived from, and "ref2" is the second reference dependency. "fd, ref1, ref2" all contain a right and a left integer representation of a subset of attributes. Note that many of these implementation are stricter than necessary, see section 3.3.3 below.

(0) Given
Input: ("fd", "ref")
Evaluates if fd is identical to ref, by checking (fd.left == ref.left && fd.right == ref.right)
(1) Reflexivity
Input: ("fd")
Evaluates if fd.right $\subseteq$ fd.left,
by checking (fd.right == fd.right & fd.left)
(2) Transitivity
Input: (fd, ref1, ref2)
Evaluates if fd is a "transitivity shortcut" $A \rightarrow C$ between ref1 $A \rightarrow B$ and ref2 $B \rightarrow C$
by checking (fd.left == ref1.left and ref1.right == ref2.left and fd.right == ref2.left).
Also we made this function accept if ref1 and ref2 order are swapped, so if ref2 is $A \rightarrow B$ and ref1 is $B \rightarrow C$.
(3) Augmentation
Input: (fd, ref)
Evaluates if fd is an augmentation of ref, ex: if ref is $A \rightarrow B$ and fd is $AC \rightarrow BD$ where $D \subseteq C$. We check whether (ref.left $\subseteq$ fd.left and ref.right $\subseteq$ fd.right and (fd.right $\setminus$ ref.right $\subseteq$ fd.left) The binary operations for this are depicted in figure 1 using php.
(4) Union
Input: (fd, ref1, ref2)

```php
function AugmentationCheck($fd, $ref1){
    $r1_subeq_rx = $ref1->right == ($ref1->right & $fd->right);
    $l1_subeq_lx = $ref1->left == ($ref1->left & $fd->left);
    $difference = $fd->right ^ $ref1->right;
    $diff_subeq_lx = $difference == ($difference & $fd->left);
    return $r1_subeq_rx && $l1_subeq_lx && $diff_subeq_lx;
}
```

Figure 1: Augmentation in PHP

```php
function PseudoTransitivityCheck($fd, $ref1, $ref2){
    $condition1a = $ref1->right == ($ref1->right & $ref2->left);
    $condition2a = $fd->left == ($ref1->right ^ $ref2->left | $ref1->left);
    $condition3a = $fd->right == $ref2->right;

    $condition1b = $ref2->right == ($ref2->right & $ref1->left);
    $condition2b = $fd->left == ($ref2->right ^ $ref1->left | $ref2->left);
    $condition3b = $fd->right == $ref1->right;
    return $condition1a && $condition2a && $condition3a || $condition1b && $condition2b && $condition3b;
}
```

Figure 2: Pseudo-Transitivity in PHP

Evaluates if (fd.left == ref1.left == ref2.left) and (fd.right == ref1.right ∪ ref2.right) so using binary operations, that is (fd.right == (red1.right | ref2.right))

(5) Decomposition
Input: (fd, ref)
Evaluates if (fd.left == ref.left) and (fd.right ⊆ ref.right) i.e. if (fd.right == fd.right & ref.right)

(6) Pseudo-Transitivity
Input: (fd, ref1, ref2)
Evaluates if (ref1.right ⊆ ref2.left) and (fd.left == (ref2.left \ ref1.right) ∪ ref1.left), and fd.right == ref2.right). The function also accepts if ref1 and ref2 are swapped. The resulting function php is displayed in figure 2

3.3.3 *Decisions on Strict Definitions of Steps.*

The augmentation and reflexivity axioms of functional dependencies lead to larger encompassing rule that if the reference $X \rightarrow Y$ is true then $W \rightarrow Z$ holds, where $X \subseteq W$ and $Z \subseteq Y \cup W$, in other words, we may always add attributes to the left, may always remove attributes from the right, and may always add attributes from the left to the right as well. Framing the inference rules to implicitly understand this, we see that transitivity checks are a special case of pseduo-transitivity checks, where the "additional middle" attribute set happens to be empty, and that reflexivity checks are a special case of augmentation checks where the pre-existing attribute set is empty. Similarly, a union check can be generalized to that if one fd's left is a subset of the other then so long as the larger left side is used, then the union holds. Taken this way, Augmentation is a special case of the generalized Union check. Finally Decomposition steps are always optional, and are can be trivially included within the other checks without need to mention them. In the interest of teaching armstrong's axioms in an atomic manner, our implementation actually uses a stricter definition of some of the inference rules, generally avoiding empty-set cases of pseudo-transitivity and augmentation. An example would be that be showing that A→ BC, C→ D implies AE → D takes 3 steps minumum.

3.3.4 *Operations for Attribute Closure Checks.*

Grading the Attribute Closure question type uses a similar approach as the Logical Consequence Problems, but with the major difference of a separate submission format. Where each step of the Logical Consequence submissions contained a functional dependency a rule, and some references, the steps to an Attribute Closure submission include a single set, and optionally a rule or reference.

The only rules that are accepted for Attribute closure steps are: Reflexivity, Given, and an empty rule input.

The behavior for Reflexivity is as follows: If the newly submitted set is the union of the previous set, and the given attributes that define the closure, then we accept, otherwise the check fails. Also reflexivity does not allow for 1 or more references, there must be no dependencies referenced. Given and empty, with no references attached, also accept the same as Reflexivity. This check reads as (newset == (oldset | Attributes))

The behavior for the "Given" rule or the empty rule input, and exactly one reference is as follows:

If the left side of the reference dependency is a subset of the previous set, and the new set is the union of the previous set and the right side of the dependency, then accept the check. (ref.left == ref.left & oldset) && (newset == oldset | ref.right)

If the submission has more than one reference per step, it is rejected as a formatting failure. We can continue in this way to ensure that each step of the attribute closure calculation is shown.

4 DATABASE DESIGN IMPLEMENTATION

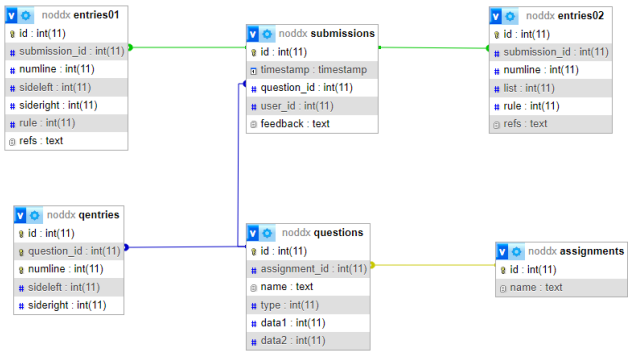

Figure 3: Database Scheme

The schema for the noddx database is designed to support the storage and organization of various types of data related to submissions, questions, assignments, and feedback. The

database consists of six tables, each with a specific purpose and relationships with other tables.

4.1 noddx_entries01

The noddx_entries01 table contains information related to individual entries for questions concerning the logical consequence of F. The information in this table includes is an id to uniquely identify each entry, submission_id to link to the submission it belongs to, numline to denote the line number of each entry of the students proof based answer, sideleft and sideright to specify the left and right side decimal values where their binary equivalent represent functional dependencies the student is using in their entry, and rule to identify the rule associated with the entry such as "Given" or one of Armstrong's Axioms. Each rule is encoded as an integer. Additionally, the table includes a column called refs which stores textual references which correspond to the numlines of other entries in table noddx_entries01 that are a part of the same submission as the current entry or entries in table noddx_qentries which is initial given entries that are a part of the same submission. The entries in this table can contain either zero, one, or two references where for zero references it is blank, for one reference it is "X", and for two references it is "X, Y" where X and Y are integers in strings which are referring to numlines for entries in the proof either for noddx_entries01 or noddx_qentries.

4.2 noddx_qentries

The entries in the table noddx_qentries represent the initial information that the student is given to create their proof when responding to the question asked. The noddx_qentries table is similar to noddx_entries01 but is specifically for question entries. It includes an id to uniquely identify each entry, question_id to link to the corresponding question, numline to denote the line number, sideleft and sideright to specify the left and right side decimal values where their binary equivalent represent functional dependencies.

4.3 noddx_submissions

The noddx_submissions table is designed to capture submission-related data, with fields for id, timestamp for the time the submission was made, question_id to link the submission to one of the specific questions, user_id to link to the user who made the submission, and feedback to store textual feedback for the submission which will be displayed to the user after their submission has been graded.

4.4 noddx_questions

The noddx_questions table contains information about questions, including an ID to uniquely identify each question, assignment_id to link to the assignment it belongs to, name to store the name of the question, type to specify if the question is a logical consequence of F question or an attribute closure question, and data1 and data2 to store additional data related to the questions where data1 is the final left side of the functional dependency which should be proved for the logical consequence questions or it is the initial set for the attribute closure question and data2 is the final right side of the functional dependency which should be proved for the attribute closure questions.

4.5 noddx_entries02

The noddx_entries02 table is similar to noddx_entries01 but with the attribute list instead of sideleft and sideright in the noddx_entries01 table; list should be a decimal value whose binary equivalent represents the current set of attributes in our attribute list. The table like the noddx_entries01 table also has the attributes submission_id, numline, rule, and refs all with very similar implementations. It should be noted that any rule can be inputted for the attribute rule but "Given" and "Reflexivity" are correct for the first entry of the submission and "Given" or "Null" (nothing) should be used as the rule for every entry after the first entry in the same submission. The attribute refs should always have one reference which functions identically to the reference in noddx_entries01.

4.6 noddx_assignments

Lastly, the noddx_assignments table contains information about assignments, including an id to uniquely identify each assignment and a name to store the name of the assignment.

4.7 Relationships in the noddx database

The schema makes use of primary and foreign key constraints to establish relationships between tables which helps in ensuring data integrity and facilitating efficient querying. The use of integer data types for IDs and references to other entities provides a standardized and efficient method for uniquely identifying and linking records. Also, the inclusion of text data types allows for flexibility in storing textual information such as feedback and references.

4.8 Conclusions

The schema we designed is supposed to provide a complete and structured framework for storing and managing the diverse data elements related to entries, submissions, questions, and assignments in the noddx database.

5 DISPLAY DESIGN FEATURES

Our project was primarily developed on the backend. However, we still needed an effective way to display the results of the student submissions being graded by our algorithm.

5.1 Display Design of the Logical Consequence of F Submissions

The display for the logical consequence of F questions showcases the evaluation results for student answers in the format of the question prompt, student submissions, and the resulting evaluation. Each submission is labeled with a unique Submission#X (where X is the submission_id), followed by the individual Entry, Justification, and Validity status. The results are broken down into the Given portion, which lists the initially given information, and the Student Entries, which represent the student's responses. The final evaluation verdict for each submission is provided under the Result section along with any extra feedback we could return to the student.

5.2 Display Design of the Attribute Closure Submissions

As for the attribute closure questions, the display structure follows a similar format, with the Submission#X (where X is the submission_id) label, Entry, Justification, and Validity sections. The Given portion outlines the initial information presented, and the Student Entries showcase the student's input. The evaluation result of whether the student is correct or not, and any extra descriptive information we could return, is presented under the Result section.

5.3 Display Design Examples

Let's look at some example results of a submission to a logical consequence of F question after the submission has been evaluated.

5.4 Correct Submission Example

Question#1

Consider the relational schema R(A,B,C,D,E) with the set F of functional dependencies F = {A->B, B->C, A->D, CD->AE}. Show the logical steps needed to prove BD->AC.

Submission#1

#	Entry	Justification	Valid?
1	A⟶B	Given	
2	B⟶C	Given	
3	A⟶D	Given	
4	CD⟶AE	Given	
	Student entries		
5	B⟶C	Given 2	✓
6	CD⟶AE	Given 4	✓
7	CD⟶ACE	Augmentation 6	✓
8	BD⟶ACE	PseudoTransitivity 5, 7	✓
9	BD⟶AC	Decomposition 8	✓

Result:

Correct!

Figure 4: Correct Submission

Looking at figure 4, the student's answer and the logical steps to prove BD->AC are displayed on the webpage. The submission has been marked as "Correct," indicating that the algorithm has recognized the student's response as accurate. The student's logical steps to prove BD->AC are as follows:

(1) B->C Given 2
(2) CD->AE Given 4
(3) CD->ACE Augmentation 6
(4) BD->ACE Pseudo Transitivity 5, 7
(5) BD->AC Decomposition 8

In addition, there is a "Valid?" column showing "✓" for each step, confirming the correctness of the logical steps taken by the student. The evaluation algorithm has successfully verified the student's submission containing the logical steps needed to prove the functional dependency as correct.

5.5 Incorrect Submission Example

Question#5

Consider the schema R=(A,B,C,D,E,F) with the set F of functional dependencies F={AB->C, BC->AD, D->E,CF->B}. Find the attribute closure {A,B}+

Submission#15

#	Entry	Justification	Valid?
1	AB⟶C	Given	
2	BC⟶AD	Given	
3	D⟶E	Given	
4	CF⟶B	Given	
	Student entries		
	{AB}+ = {AB}	Reflexivity	✓
	{AB}+ = {ABC}	Given 1	✓
	{AB}+ = {ABCD}	Given 2	✓
	{AB}+ = {ABCDE}	Given 3	✓
	{AB}+ = {ABCDEF}	Given 4	✗

Result:

Incorrect. There's a problem with the logic.

Figure 5: Incorrect Submission

In figure 5, we can see the student has provided their answer and justification, and the evaluation algorithm has determined the result to be an incorrect response with a problem in the logic. The student entry for attribute closure in the given functional dependency has been evaluated step by step, and the evaluation result shows where the student's logic has gone wrong. Based on the evaluation, the attribute closure for the final attribute set by the student seems to have been incorrectly calculated, as indicated by the checkmark under "Justification Valid?" and the "X" in the "Result" column. The evaluation algorithm has identified the problem in the logic, as indicated by the note "Incorrect. There's a problem with the logic." which makes sense since the "Given 4" justification corresponds to the 4th entry which contains the functional dependency "CF->B" which can not be used to add F to the current attribute set. It should be noted that no more of the proof is displayed after an incorrect step as it is difficult to determine if the succeeding proof steps are impacted by the incorrect step, and ultimately there is the same result of an incorrect proof. If the student wants to see if the steps after an incorrect step are correct, they can fix the incorrect step in their proof and reevaluate their answer.

5.6 Display Design Conclusion

Both displays effectively organize the student submissions, the corresponding justifications and validity assessments of

each submission, and the final evaluation result of each question, providing a clear and structured overview of the evaluated proofs for each type of question.

6 TESTING AND EVALUATION

From all the testing we have performed, we are confident that our algorithms will perform as desired. For one, we believe that our application of the underlying theory is sound. To be sure of it, we have performed extensive testing. We tried to perform a variety of tests where our answers highlighted different applications of the theory in order to catch as many edge cases as we could. Some of these tests have included cases provided by our instructor as well as cases designed by us. If you are not yet confident of our algorithm implementation, it is easy to create and perform your own tests. To do so the data should first be entered into the database, and then on the web page interface, you can perform the tests of the cases entered the database. From there, you can visually evaluate if our results align with what they should actually be.

7 FUTURE IMPROVEMENTS

Future improvements to the project could focus on enhancing the grading and feedback mechanisms to provide a more comprehensive and personalized learning experience for students. In addition, the project could be expanded so that more question types could be graded. Here are the key enhancements that could be made explained in more detail:

7.1 Better Grading

A more detailed oriented grading system could be implemented that goes beyond the current binary correct/incorrect assessment. The system should allow teachers to define grading criteria on a scale, such as out of 10, and take into account factors like the number of correct steps versus incorrect steps made by the student. The goal would be to provide a more accurate and fair evaluation of the student's understanding and performance.

7.2 Better Feedback

7.2.1 Personalized Feedback.

More detailed feedback on incorrect student answers can be provided, including specific explanations of why the answer is wrong. This personalized feedback would help students understand their mistakes and learn from them, ultimately improving their grasp of the subject matter.

7.2.2 Complete Feedback.

Currently, if a step in a student's proof is wrong, the evaluation system does not grade after the incorrect step. The evaluation algorithm could be improved so that it grades every answer after the first incorrect step. A design decision would need to be made on whether to mark a subsequent step which references and incorrect step as correct because the logic it uses is correct even though it uses an incorrect step as a reference.

7.3 Hinting System

Another useful feature that could be implemented is a hinting system which guides students towards the correct answer. This feature should be customizable, allowing instructors to enable or disable it based on their teaching approach. The hints might also alter the way the assignment is graded if used by the student; this feature would again ideally have the ability to be enabled or disabled by the instructor.

7.4 Grading More Question Types

7.4.1 Candidate Keys.

In the future, the evaluation algorithm can be expanded to include the grading of student proofs for finding candidate keys. The kinds of candidate key proofs that could be graded include the exhaustive, heuristic, or tree method. This would provide a more comprehensive evaluation of the students' understanding of the concept.

7.4.2 Normal Form Decomposition.

Another potential improvement that can be made to the project is the introduction of a feature to check if the student correctly decomposed a database schema into the third normal form or Boyce-Codd normal form. Because checking if the student performed the normal form decomposition correctly requires finding a canonical cover of the initial set of functional dependencies and the appropriate candidate keys that correspond to it, the functionality of this feature likely involves the functionality of being able to check if a functional dependency is a logical consequence of a set of functional dependencies, being able to find the correct attribute closure given a set of functional dependencies and an initial set of attributes, and being able to find all of the correct candidate keys of a set of functional dependencies. Implementing this feature to the evaluation system would provide a more thorough assessment of the students' grasp of database normalization concepts.

8 CONCLUSION

My partner and I take pride in the accomplishments we have achieved. Our innovative approach to creating evaluation algorithms for the logical consequence of F and attribute closure required conscientious thought and extensive brainstorming. Along with algorithm implementation, our design and successful querying of a tailored database showcased our technical proficiency. The binary implementation of our work was notable as it will save future groups the time we undertook in the creative design process. While we acknowledge there is always room for improvement with our project, we have

provided a project that future contributors can easily build upon; it will be interesting to see how they expand upon it. Ultimately, we think our project stands as a testament to our learning this semester.