CdbNorm: An Efficient Library for Automatic Database Normalization

Ivan Piza-Davila, Fernando Gutierrez-Preciado, Victor Ortega-Guzman, Mildreth Alcaraz-Mejia
Dept. of Electronics, Systems and Informatics, ITESO University, Guadalajara, Jal., Mexico

***Abstract*—This study introduces CdbNorm, a library that provides efficient implementations of the first three normal forms of relational database normalization. CdbNorm makes it quick and straightforward for a data analyst to divide a large dataset into smaller tables free from database anomalies (insert, update, and delete) and duplicate data. This study describes each of the steps of our normalization algorithm, which includes the discovery of functional dependencies and the population of output normalized datasets. We evaluate the accuracy and efficiency of our algorithm with databases introduced in prior papers and with large datasets available online.**

Keywords—Database normalization; functional dependency; normal form; 1NF; 2NF; 3NF

I. Introduction

In relational database design, normalization is the process of reorganizing data to minimize redundancy and thus avoid insertion, update, and deletion anomalies [13]. It is accomplished by applying several formal rules (normal forms) to the given dataset.

There have been several efforts to develop software tools that carry out the normalization of relational databases. Some of these tools [1-4] are specifically designed for teaching and learning purposes. With a friendly user interface, they provide a step-by-step guide to perform normalization of small tables, enabling users (students) to analyze the result at each step. Some of these tools keep track of user performance, while others provide practice problems that students can solve and submit. Other tools [5-9] are targeted at automating the normalization process using distinct efficient algorithmic approaches. Some of these tools include a GUI, but all of them require the user to input the database schema, which includes the table name, attributes, and the set of functional dependencies. With this data, the algorithms will then discover the primary key, candidate keys, and carry out the normalization process up to 3NF. In some cases, Boyce-Codd NF [10] is a tool that automates the process of normalization up to 3NF using only one linked list to represent a relation along with functional dependencies. This approach requires less space and time as compared with other similar works. Finally, [11] proposes a novel method that automatically provides a relational database design for an input dataset by treating database design as a multi-objective optimization problem. They use genetic algorithms to solve this problem.

In this study, we introduce a library that undertakes database normalization. Unlike most of the tools mentioned, our library accepts a non-normalized table as input and produces normalized tables that result from applying the first three normal forms on the input. All the tables are given as Tab-Separated Values (TSV) files. The output tables are properly populated with the corresponding values from the input table.

Although database normalization is well understood at the theoretical level, there is a lack of practical tools that integrate automatic functional dependency discovery from empirical data with efficient algorithmic normalization into the first three normal forms. Most existing systems require manually specified dependencies or focus only on FD discovery without generating normalized schemas or transformed datasets. Therefore, there is a need for an end-to-end, data-driven normalization framework capable of autonomously detecting dependencies and producing fully normalized outputs.

The implementation of the normalization process leverages efficient data structures, allowing it to normalize a dataset with hundreds of thousands of rows in just a few seconds. Thus, our library does not require the user to provide the database schema, particularly the functional dependencies. Instead, the algorithm discovers them automatically. When dealing with large datasets, this step can be very time-consuming for the database designer or data analyst and may lead to unattended data anomalies.

The rest of the study is organized as follows: Section II provides some theoretical background related to database normalization. Section III introduces the CdbNorm library and explains the normalization algorithms. Section IV presents some experimental results where the accuracy and efficiency of CdbNorm are tested. Finally, Section V summarizes the conclusions.

II. Background

This section introduces the foundational concepts of functional dependencies and the first three normal forms (1NF, 2NF, and 3NF), each representing a progressive refinement in the structure of relational schemas. To put these concepts in practice, we provide illustrative examples for each normal form, demonstrating how they address specific anomalies. These examples are directly tied to the implementation strategies used in CdbNorm library.

A. Functional Dependencies

A fundamental component of normalization algorithms is dependencies discovery. Given any two fields f1, f2 we say that f2 is functionally dependent on f1 or f1 determines f2 (f1 → f2) if there exist no two rows containing distinct values for f2 but the same for f1. From Table I, we can infer that *Team* determines *Country*, but not vice versa, thus the normalization process should create a catalogue of teams eventually and remove *Country* from Table I. If we removed the row with *Id* = 4000, *Team* and *Country* would determine each other, even though that is not actually true, and may add noise to the normalization process. In conclusion, an automatic dependency-discovery process would require the input table to have enough diversity in data to find what a data analyst would expect.

TABLE I. SOCCER-PLAYER TABLE

Id	*Team*	*Country*
1000	PSG	France
2000	Ajax	Netherlands
3000	Inter	Italy
4000	PSV	Netherlands

From Table II, we can notice that StudentId, StudentName, and StudentAddress determine each other, i.e., there are symmetric (or mutual) functional dependencies among all of them. In such a case, one of the fields from the group must be selected to be the one that determines the others.

TABLE II. SCHOOL-ACCESS TABLE

Id	*Time*	*StudentId*	*StudentName*	*StudentAddress*
1	July 7 2025	10	John Smith	173 Newton St.
2	July 7 2025	20	Paul Fox	85 Oak Ave.
3	July 8 2025	10	John Smith	173 Newton St.
4	July 8 2025	20	Paul Fox	85 Oak Ave.

B. *First Normal Form*

The following conditions must be met for a table to be in the First Normal Form (1NF):

1) There are no duplicate rows.
2) The attribute domain does not change.
3) Only single-valued attributes: a single cell must not hold more than one value.
4) There is a primary key for identification.

CdbNorm assumes all the rows are distinct in at least one field. Besides, the library treats all the data as String values because the normalization process is not concerned with either mathematical calculations, data validation, or datatype discovery, but with organizing data into smaller and more manageable tables free from data redundancy. Thus, CdbNorm focuses on these two tasks:

1) Find the primary key, and
2) Address multivalued attributes, i.e., cells with comma-separated values.

For the first task, we need to find a column or set of columns that uniquely identify each row of the table, i.e., no two records in the dataset contain the same data at that column. Clearly, the primary key of $SoccerPlayer$ table (Table I) is Id, whereas the primary key of $Beer$ table (Table III) is the pair $(Beer, Warehouse)$. The second one was not that clear.

TABLE III. BEER TABLE

Beer	*Brewery*	*Strength*	*City*	*Region*	*Warehouse*	*Quantity*
Choice	Websters	XX	York	North West	**1**	200
Choice	Websters	XX	York	North West	**4**	100
Choice	Websters	XX	York	North West	**8**	200
Old Bob	Websters	XXX	York	North West	**1**	300
Old Bob	Websters	XXX	York	North West	**2**	300
Landlord	Taylors	XXX	Leeds	North West	**8**	200
Landlord	Taylors	XXX	Leeds	North West	**3**	190
Directors	Fremlins	X	London	South East	**6**	400
Directors	Fremlins	X	London	South East	**4**	290
Wobbly Joe	Sam Smith	XXXX	York	North West	**3**	90
Watery	Whibread	X	London	South East	**5**	300

TABLE IV. NAMEBASICS TABLE

nconst	*Primary name*	*Birth year*	*Death year*	*Primary profession*	*Known for titles*
nm0000001	Fred Astaire	1899	1987	actor, producer, miscellaneous	tt0072308, tt0050419, tt0027125, tt0031983
nm0000002	Lauren Bacall	1924	2014	actress, soundtrack, archive_footage	tt0037382, tt0075213, tt0117057, tt0038355
nm0000003	Brigitte Bardot	1934	--	actress, producer, music_department	tt0057345, tt004918, tt0056404, t0054452

TABLE V. PRIMARYPROFESSION AND KNOWNFORTITLES TABLES

nconst	*Primary profession*		*nconst*	*Known for titles*
nm0000001	**actor**		**nm0000001**	**tt0072308**
nm0000001	**producer**		**nm0000001**	**tt0050419**
nm0000001	**miscellaneous**		**nm0000001**	**tt0027125**
nm0000002	**actress**		**nm0000002**	**tt0031983**
nm0000002	**soundtrack**		**nm0000002**	**tt0037382**
nm0000002	**archive_footage**		**nm0000002**	**tt0075213**
:	*3 more rows*		:	*6 more rows*

Regarding the second task, every column containing comma-separated values will be treated as a multivalued attribute (mva). For each mva found, there will be an output table. The 1NF will pair each value in the comma-separated list with the primary key in that row and add the pair into the output table corresponding to the current mva. Table IV (NameBasics) contains two multivalued attributes: PrimaryProfession and KnownForTitles. As a result of the 1NF, these two fields will be removed from the table, and two new tables will be created (see Table V), one for each mva, containing the result of pairing the primary key (nconst) with every value in the comma-separated list at the same row. Notice that all the fields in these two tables make up the primary key.

C. Second Normal Form

The second Normal Form (2NF) applies only to tables with a composite primary key. The following must be met for a table to be in 2NF: every non-primary-key attribute is fully functionally dependent on the primary key, not on any proper subset of it. Thus, SoccerPlayer (Table I) and NameBasics (Table IV) are automatically in 2NF because the primary key is a single field, whereas Beer (Table III) requires further analysis: we can see that fields Brewery, Strength, City and Region are all functionally dependent on the primary-key field Beer. Therefore, the 2NF will remove these fields from the main table (see Table VI) and create a catalogue table having Beer as a primary key, paired with the four fields it determines (see Table VII).

TABLE VI. BEER-WAREHOUSE TABLE IN 2NF

Beer	***Warehouse***	***Quantity***
Choice	**1**	200
Choice	**4**	100
Choice	**8**	200
Old Bob	**1**	300
Old Bob	**2**	300
Landlord	**8**	200
Landlord	**3**	190
Directors	**6**	400
Directors	**4**	290
Wobbly Joe	**3**	90
Watery	**5**	300

TABLE VII. BEER CATALOGUE IN 2NF

Beer	***Brewery***	***Strength***	***City***	***Region***
Choice	Websters	XX	York	North West
Old Bob	Websters	XXX	York	North West
Landlord	Taylors	XXX	Leeds	North West
Directors	Fremlins	X	London	South East

D. Third Normal Form

A table is in the third Normal Form (3NF) if no non-primary-key field is transitively dependent on the primary key. This implies that no functional dependency $A \rightarrow B$ between any two non-primary-key fields A, B must exist. In such a case, for every $A \rightarrow B$ found, the 3NF process will remove B from the table and create a new table "Catalog_A", having A as a primary key, and storing all the distinct pairs (A, B). If it is found that A defines another field C as well, then a new table will not be created but instead, the 3NF process will add column C to "Catalog_A" and fill it with the corresponding data. Ultimately, this table will contain all the distinct tuples (A, B, C) found in the input table.

From the Beer catalogue (Table VII), we can note that Brewery determines City, and Region is functionally dependent on City. Therefore, all the determined fields (City, Region) will be removed from Beer catalogue, and two new catalogues are created: the first one to pair each distinct Brewery to its city, and the other to pair each distinct City with its Region (see Table VIII and Table IX).

TABLE VIII. BEER CATALOG IN 3NF

Beer	***Brewery***	***Strength***
Choice	Websters	XX
Old Bob	Websters	XXX
Landlord	Taylors	XXX
Directors	Fremlins	X
Wobbly Joe	Sam Smith	XXXX
Watery	Whibread	X

TABLE IX. BREWERY AND CITY CATALOGUES IN 3NF

Brewery	***Primary profession***		***City***	***Region***
Websters	York		**York**	North West
Taylors	Leeds		**Leeds**	North West
Fremlins	London		**London**	South East
Sam Smith	York			
Whibread	London			

III. Normalization Algorithms

The normalization process carried out by *CdbNorm* follows a sequence of steps that will be described in this section. These steps range from loading an input dataset (TSV file) to creating output TSV files containing a normalized populated table.

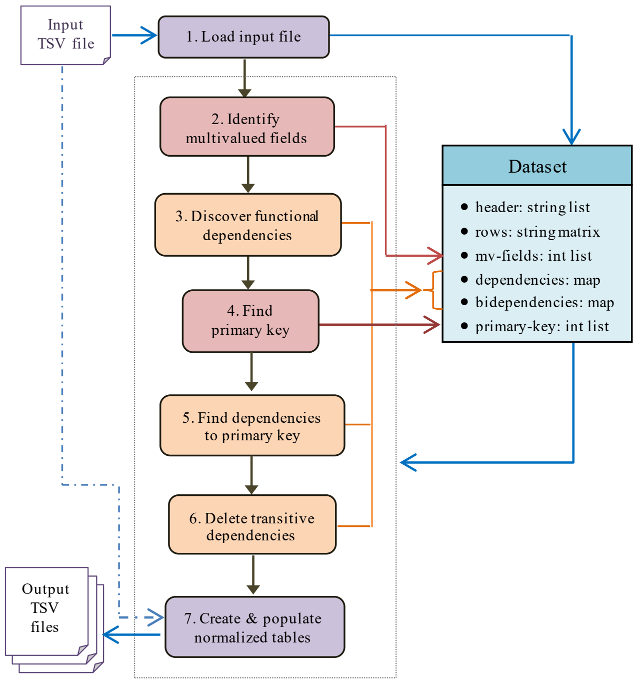

Fig. 1. Flowchart for CdbNorm normalization algorithm.

The flowchart in Fig. 1 depicts the overall normalization algorithm of CdbNorm. We can see that the first step, Load dataset, produces a Dataset structure that will be used throughout the rest of the normalization steps. This structure not only stores the headers and the rows of the dataset, but also the primary key and functional dependencies discovered as well as other metadata that together will be helpful to build the output normalized datasets.

Steps 2 to 6 are aimed at analyzing the available data to find a database schema that fulfils the first normal forms. They operate exclusively with data loaded into memory through Dataset structure. When dealing with huge datasets, i.e., over millions of rows of data, we can easily run into memory issues. CdbNorm allows to specify a percentage of rows to read from the dataset. In most of the cases, we don't need the entire set of rows to discover the database schema. This involves the last two steps, reading the input file while populating the output files completely.

A. Dataset Structure

The header contains the names of the fields. They are read from the first row of the TSV file, and it is primarily used for logging purposes. The rest of the rows read from the file are stored in the string matrix rows. For every column containing comma-separated values in at least one row, its index will be added to the list mv-fields (multivalued). Step 2 carries out this process. The rest of the fields of the Dataset are populated in steps 3 to 6. The field dependencies are a dictionary that maps an attribute index f to a non-empty list of indices of all the other attributes that are functionally dependent on f. Similarly, the field bidependencies maps an attribute index f to all the other attributes mutually dependent on f. Finally, the primary key is a list containing the index of all the attributes that make up the primary key.

B. Discover Functional Dependencies

The success of the normalization algorithm relies on the success of the dependencies discovery process, whereas the success of the latter depends on the diversity and correctness of the available data. Algorithm 1 returns $true$ if and only if $f \rightarrow g$, that is, there exists no two rows r, s in the dataset such that $r_f = s_f \ and \ r_g \neq s_g$. To determine this, it employs a map.

Algorithm 1: Does field *f* determine field *g*?

```
let values be a string-to-string map
For-each row in dataset.rows:
    If $row_f \in values.key$:
        if $row_g \neq values_{row_f}$
            return false
    else
        $values_{row_f} \leftarrow row_g$
            return true
```

Algorithm 2 finds all the functional dependencies of the loaded dataset, by pairing all the non-multi-valued fields and calling Algorithm 1. If a pair of fields f, g determines each other, that is, Algorithm 1 returns true with parameters (f, g) and (g, f), then the algorithm will decide that f determines g only if g was not mutually determined before. Otherwise, if Algorithm 1 returns true with only one combination of parameters, the second parameter will be added to the list of dependencies of the first one. In other words, when there exist mutual dependencies among two or more fields, the left-most located field will ultimately define all the others.

For a better understanding of Algorithm 2, consider $SchoolAccess$ dataset (Table II). All the fields will be paired with each other: Id (0), $Date$ (1), $StudentId$ (2), $StudentName$ (3), $StudentAddress$ (4). Since there are no two rows r, s such that $r[Id] \neq s[Id]$, all the fields are functionally dependent on Id and, therefore, dependencies0 = [1, 2, 3, 4]. We can see that $SubjectId$ is not functionally dependent on $Date$ because in the first two rows $StudentId$ has different values (10, 20) but $Date$ has the same (July 7 2025). Conversely, $Date$ is not functionally dependent on $SubjectId$ because in rows 1 and 3, $Date$ has distinct values (July 7, July 8) but $StudentId$ has the same (10). On the other hand, $StudentId$ and $StudentName$ both determine each other.

Algorithm 2: Find functional dependencies

```
For-each f in {1, … dataset.header.size} \ dataset.normalized:
    For-each g in { f + 1, … dataset.header.size} \ dataset.normalized:
        If $\boldsymbol{determines}(f, g) and \neq \boldsymbol{determines}(g, f)$
            If $g \notin dataset.bidependencies_h$ for all $h < g$
                $dataset.dependencies_f.add(g)$
                $dataset.bidependencies_f.add(g)$
        Else-if $\boldsymbol{determines}(f, g)$
            $dataset.dependencies_f.add(g)$
        Else-if $\boldsymbol{determines}(g, f)$
            $dataset.dependencies_g.add(f)$
```

Whenever the Id changes, the name changes accordingly, and vice versa. Since $StudentName$ was not mutually defined before, it is added to the lists of dependencies and bidependencies of $StudentId$. A similar situation will happen between $StudentId$ and $StudentAddress$. The last pair to be processed is $(StudentName, StudentAddress)$. Since they both determine each other but $StudentAddress$ was mutually defined before (with $StudentId$), then no dependency will be added here. At the end, we will get the following functional dependencies:

1) $Id \rightarrow Date, StudentId, StudentName, StudentAddress$
2) $Student\_Id \rightarrow StudentName, StudentAddress$

If we run Algorithm 2 with Beer dataset (Table III), we will get the following dependencies. Notice that no mutual functional dependencies exist in that table.

1) $Beer \rightarrow Brewery, Strength, City, Region$
2) $Brewery \rightarrow City, Region$
3) $Strength \rightarrow Region$
4) $City \rightarrow Region$
5) $Quantity \rightarrow Region$

C. Find Primary Key

CdbNorm seeks the left-most ordered subset of fields of minimal size such that there do not exist two rows containing respectively the same values at all the columns from the subset, and there are no functional dependencies between any two members of the subset. To achieve this, it produces combinations of non-multivalued field indices or selectable fields (see Algorithm 3). Each combination is a candidate key and ranges from an individual field to the entire set of fields. The algorithm stops when a combination for which no two rows share the same values is found.

If we run Algorithm 3 with the Beer dataset (Table III), it will first create the following combinations to find the primary key: [Beer], [Brewery], [Strength], [City], [Region], [Warehouse], [Quantity]. For the first five combinations, the second row will lead to duplicate values. Then, it will pair Beer with Brewery, Strength, City and Region, leading to duplicates on the second row as well. The next combination [Beer, Warehouse] will finally lead us to distinct values throughout the dataset and, therefore, it will be set as the primary key.

Algorithm 3: Find primary key

```
Input: selectable: list of all non-multivalued fields
       candidate: a list containing the first non-multivalued field
Let values-set be a set of string lists
Let found-duplicates ← false
For-each row in dataset.rows:
        | Let values be the list {$row_f$ | $f \in candidate\_key$}
        | If values ∈ values-set
        |       | found duplicates ← true
        | values-set ← values-set ∪ {values}
If found-duplicates
    | candidate ← next-combination(selectable, candidate)
    | find-primary-key (selectable, candidate)
Else
    | set-primary-key (candidate-key)
```

D. Find Dependencies to Primary Key

If the primary key found is made up of only one field, this step is unnecessary because its dependencies were already discovered two steps before. As a result of this step, our composite primary key will determine all the non-key fields that are not functionally dependent on a key field (see Algorithm 4).

Algorithm 4: Find dependencies to primary key

```
If size (key) = 1
    | return
Let banned-fields be a list of integers
For-each (k, list) in dataset.dependencies:
    | If k in dataset.primary-key
    |     | For-each g in list:
    |     |     | banned-fields.add (g)
For-each (k, list) in dataset.dependencies:
    | If f not in (dataset.primary-key ∪ banned-fields)
    |     | dataset.dependencies$_{-1}$.add(f)
```

The list of such fields will be stored in dependencies map with key = –1. At the end of the normalization process, these fields will be removed from the main table in compliance with the second normal form. If we take Beer table as an example, the dependencies to primary key (Beer, Warehouse) includes only the field Quantity because Brewery, Strength, City, Region are all functionally dependent on key field Beer.

E. Delete Transitive Dependencies

The third normal form must ensure that every non-key attribute in the dataset is directly dependent on the primary key and not indirectly dependent through another non-key attribute, which would otherwise lead to transitive dependencies. Therefore, in this step all the transitive dependencies must be identified and removed.

Algorithm 5 shows the way transitive dependencies are found through a queue of determiner attributes, starting from the primary key, followed by the attributes that make up the key, if composite. For a better understanding of the algorithm, let us consider the simplest form of a transitive dependency: $A \rightarrow (B, C), B \rightarrow C.$ Clearly, there exists a transitive dependency between A and C. In this case, the primary key must be A. Why not B? Because no other attribute would determine A, and step 5 of the normalization process, find dependencies to primary case, should have handled this case. So, A is initially added to the queue and is the first element to be popped out the queue. A has two dependencies: B and C. B is not determined by any other, so it is pushed into the queue; but C is determined by B, so it is removed from the list of dependencies of A. Then, B is popped out, and C is pushed into the queue. Finally, C is removed from the queue.

Algorithm 5: Delete transitive dependencies

```
Let determiners be a queue of integers
If size(dataset.primary-key) > 1
|       determiners.push(-1)
For-each f in dataset.primary-key:
|       determiners.push(f)
While determiners is not empty:
|       Let toRemove be a set of integers
|       Let f ← determiners.pop()
|       For-each d in dataset.dependencies$_f$
|       |   If $d \in dataset.dependencies_g$ for some $g \neq f$
|       |   |       | toRemove.add(d)
|       |   Else definers.push(d)
|   For-each r in toRemove
|       | dataset.dependencies$_f$.remove(r)
|   If $f \geq 0$ and dataset.dependencies$_f$ is empty
|       | dataset.dependencies.remove(f)
```

Taking Beer table as an example, the queue will contain at first three elements: the composite primary key (Beer, Warehouse) and its two attributes separately. The first element to be popped out of the queue will be the primary key. This element will add Quantity to the queue. The evolution of the query and the way the functional dependencies are changing all throughout the algorithm are shown in the listing below that starts when Beer is removed from the queue. Since City and Region are both determined by others they are removed from Beer's dependencies.

1) $Beer \Leftarrow [Warehouse, Quantity] \Leftarrow Brewery, Strength$
2) **$Beer \rightarrow Brewery, Strength,$ ~~$City, Region$~~**
3) $Warehouse \Leftarrow [Quantity, Brewery, Strenth]$
4) $Quantity \Leftarrow [Brewery, Strength]$
5) **$Quantity \rightarrow$ ~~$Region$~~**
6) $Brewery \Leftarrow [Strength] \Leftarrow City$
7) **$Brewery \rightarrow City,$ ~~$Region$~~**
8) $Strength \Leftarrow [City]$
9) **$Strength \rightarrow$ ~~$Region$~~**
10) $City \Leftarrow [\ ] \Leftarrow Region$
11) $Region \Leftarrow [\ ]$

F. Create and Populate Normalized Tables

For each multivalued attribute found in the dataset, a new TSV file is created. This file contains the fields that make up the primary key discovered and the multivalued attribute. All these fields will compose the primary key of this new table. If the input file was loaded into the dataset structure entirely as part of step 1, the input file is not revisited to create the output files. Otherwise, the algorithm reads the input file as it populates the new table. For each row and each value in the comma-separated list of the multivalued attribute, a new row is inserted into the output file, containing the primary key and the value.

For each entry (d, list) in dependencies map, a new TSV file is created. This file contains the attribute d as its primary key, and all the attributes contained in list. If list is empty, this entry is skipped unless d is the primary key. As in the previous step, if the input file was not loaded into memory entirely, the algorithm needs to read the input file to populate the table. For each row from the input file, a new row containing the values at columns d, list1, list2,…,listn is added to the output file, only if such combination of values is unique. A set is employed to handle duplicates.

IV. Experimental Results

All the experiments presented in this section were conducted on a portable computer running Windows 10 Enterprise, equipped with an Intel Core i7-1165G7 processor operating at 2.80 GHz, and 32 GB of RAM.

To test the accuracy of our normalization algorithms, we have collected six standard relations from various database books and research papers that are used to explain normalization up to 3NF. All these relations are described and benchmarked in [10]. We have selected these relations because, despite having few rows, they all pose challenges for our normalization algorithm, namely: composite keys, partial and full dependencies to composite keys, transitive and mutual dependencies, as can be seen in Table XI.

The TSV files can be found in our repository (github.com/hpiza/cdbnorm/tree/main/datasets). Table X shows the number of rows and attributes of each relation.

TABLE X. Description of Relations used for Testing Accuracy

Relation	Attributes	Rows
Beer	beer, brewery, strength, city, region, warehouse, quantity (see Section 2)	11
ClientRental	clientNo, propertyNo, cName, pAddress, rentStart, rentFinish, rent, ownerNo, oName	8
Invoice	Order_ID, Order_Date, Customer_ID, Customer_Name, Customer_Address, Product_ID , Product_Description, Product_Finish, Unit_Price, Order_Quantity	8
Report	reportNo, editor, deptNo, deptName, deptAddress, authourId, authourName, authourAddress	8
StaffProperty	PropertyNo, idate, itime, pAddress, coments, staffNo, sName, carReg	7
Wellmeadows	Patient_No, DrugNo, Start_Date, Full_Name, Ward_No, Ward_Name, Bed_No, Drug_Name, Description, Dosage, Method_Admin, Units_Day, Finish_Date	10

TABLE XI. Results of Running CdbNorm with Standard Relations.

Relation	Primary key	Functional dependencies	t1	t2	M
Beer	beer, warehouse	beer, warehouse → quantity beer → brewery, strength brewery → city city → region	2	11	4.1
ClientRental	clientNo, propertyNo	clientNo, propertyNo → rentStart, rentFinish clientNo → cName propertyNo → pAddress, rent, ownerNo ownerNo → oName	1	6	4.2
Invoice	orderId, productId	orderID, productID → orderedQuantity orderID → orderDate, customerID customerID → customerName, customerAddres productID → productDescription, productFinish, productStandardPrice	1	7	4.6
Report	reportNo, authourId	reportNo, authorId → {} reportNo → editor, deptNo deptNo → deptName, deptAddr authorId → authorName, authorAddr	1	5	3.5
StaffProperty	propertyNo, iDate	propertyNo, iDate → iTime, comments, staffNo, carReg propertyNo → pAddress staffNo → sName	1	6	3.2
Wellmeadows	patient-Number, drugNumber, startDate	patientNumber, drugNumber, startDate → wardNumber, unitsPerDay, finishDate patientNumber → fullName wardNumber → bedNumber, wardName drugNumber → name, description, dosage, methodOfAdmin	3	8	6.9
Average			**1.5**	**7.2**	**4.4**

Table XI shows the primary key and functional dependencies found for each relation. Each functional dependency led to the creation of an output populated table (TSV file) which primary key is the left-hand side of the dependency. We can see that all the output tables match those in the experimental results of [10]. The last 3 columns of this table t1, t2, m denote respectively the time in milliseconds that took the normalization steps (2 to 6) exclusively, the time in milliseconds that took the entire process (steps 1 to 7) and the memory required in kilobytes. All these datasets were entirely loaded into memory. Even though none of these datasets have multivalued fields, they all have composite keys, partial dependencies to the primary key, and transitive dependencies that must be addressed. We can see that the average time to perform the entire process of normalization and table creations was 7.2 milliseconds, whereas the average time to process these six datasets by RDBNorm [10] was 998 milliseconds. The average memory allocated by CdbNorm was 4.4 kilobytes, whereas RDBNorm required 6.9 kilobytes in average.

To challenge CdbNorm efficiency in real-life like scenarios, we ran the normalization algorithms with seven large public IMDb datasets, which are described in [12]. As can be seen in Table XII, each of these datasets is made up of millions of rows. The size of the files ranged from 26 megabytes to 4 gigabytes.

TABLE XII. DESCRIPTION OF IMDB DATASETS

Time	Attributes	Rows	File size
NameBasics	nconst, primaryName, birthYear, deathYear, primaryProfession, knownForTitles	14,461,634	868,230
TitleAkas	titleId, ordering, title, region, language, types, attributes, isOriginalTitle	52,526,985	2,582,130
TitleBasics	tconst, titleType, primaryTitle, originalTitle, isAdult, startYear, endYear, runtimeMinutes, genres	11,762,395	993,177
TitleCrew	tconst, directors, writers	11,764,260	378,877
TitlePrincipals	tconst, ordering, nconst, category, job, characters	93,482,991	4,068,502
TitleEpisode	tconst, parentTconst, seasonNumber, episodeNumber	9,054,578	231,782
TitleRatings	tconst, averageRating, numVotes	1,586,591	26,971

Table XIII shows the results of running our normalization algorithms on each of the large datasets, which includes primary key discovered, functional dependencies found, multivalued attributes identified, and size of the output tables expressed as columns x rows. Unlike the standard relations, most of these datasets include at least one multivalued attribute. On the other hand, none of them required to address partial dependencies to primary key, and only one table (TitleAkas) had transitive dependencies.

We can see that for each multivalued attribute and functional dependency a new output table was created. For example, the largest dataset, TitlePrincipals, required the creation of three output tables containing over 93 million of rows, and the following attributes each: 1) tconst, ordering, nconst, category, 2) tconst, ordering, job, 3) tconst, ordering, characters. In all cases, the primary key was the pair <tconst, ordering>.

Table XIV shows the running time that took CdbNorm to process each of the large datasets. Columns t1, t2 and m denote, respectively, the time in seconds that took the normalization steps, the time in seconds that took the entire process, and the memory required in megabytes. For each dataset, we ran the normalization process four times, for each time, we loaded a different percentage of the file: 100%, 10%, 1%, 0.1%. In all cases, the normalization algorithm discovered the same primary key, functional dependencies and multivalued attributes.

TABLE XIII. RESULTS OF RUNNING CDBNORM WITH IMDB DATASETS

Time	Primary Key	Functional dependencies	Multivalued attributes	Size of output tables
Name-Basics	nconst	nconst → birthYear, deathYear	primary-name, primary-profession, known-forTitles	3 x 14'461,634 2 x 14'461,720 2 x 18'934,493 2 x 25'282,280
TitleAkas	titleId, ordering	titleId, ordering → region, language, types, attributes types → isOriginalTitle	title	6 x 52'526,985 2 x 24 3 x 53'222,103
Title-Basics	tconst	tconst → titleType, isAdult, startYear, endYear, runtimeMinutes	primaryTitle, originalTitle, genres	6 x 11'762,395 2 x 12'199,419 2 x 12'198,927 2 x 18'797,502
TitleCrew	tconst	None	directors, writers	2 x 13'971,863 2 x 19'735,108
Title-Principals	tconst, ordering	tconst, ordering → nconst, category	job, characters	4 x 93'482,991 3 x 93'499,152 3 x 93'808,896
Title-Episode	tconst	tconst → parentTconst, seasonNumber, episodeNumber	None	4 x 9'054,578
Title-Ratings	tconst	tconst → averageRating, numVotes	None	3 x 1'586,591

TABLE XIV. EFFICIENCY OF CDBNORM PROCESSING IMDB DATASETS

	t1	**t2**	**M**	**t1**	**t2**	**M**
Rows loaded	**100%**			**10%**		
NameBasics	94	175	4024	7.8	135	403
TitleAkas	266	502	19165	31	363	1917
TitleBasics	198	285	4905	13	136	491
TitleCrew	43	80	1656	3.3	47	165
TitlePrincipals	**567**	1021	**25278**	**50**	658	**2527**
TitleEpisode	77	97	1623	5.5	85	162
TitleRatings	9	12	213	0.7	3.4	21
	t1	t2	M	t1	t2	M
Rows loaded	**1%**			**0.1%**		
NameBasics	0.7	121	40	0.13	99	4.0
TitleAkas	2.0	342	192	0.24	338	19.4
TitleBasics	1.1	123	49	0.11	133	4.9
TitleCrew	0.3	44	16	0.03	45	1.7
TitlePrincipals	**2.7**	574	**253**	**0.25**	559	**25.6**
TitleEpisode	0.5	79	16	0.05	83	1.6
TitleRatings	0.07	2.6	2.1	0.01	3.1	0.2

We can notice that the normalization time and the amount of memory required to process the large dataset (TitlePrincipals) decreased proportionally to the percentage of rows loaded: <100%, 567, 25 GB>, <10%, 50, 2.5 GB>, <1%, 2.7, 253 MB> and <0.1%, 0.25, 25.6 MB>, whereas the total running time was not affected at all by the fact that the input file was revisited to create the output normalized datasets (1021, 658, 574, 559).

V. CONCLUSION AND FUTURE WORK

In this study, we presented in detail CdbNorm, an efficient C++ based library useful to normalize a table up to the Third Normal Form. Unlike most state-of-the-art normalization tools, our library takes in the unnormalized data, instead of the database schema. The process includes discovery of multivalued attributes, functional dependencies, transitive dependencies and the primary key. With this information, CdbNorm creates the normalized tables which are populated according to the input data. Several experiments proved that the algorithm outperforms state-of-the-art algorithm and behaves very well when dealing with large datasets. Finally, since CdbNorm is publicly available from our repository: https://github.com/hpiza/cdbnorm. It can serve as a common base for other researchers aiming to improve or extend our normalization algorithms, and for benchmarking purposes.

Future work for this library includes: 1) improving the selection of defining attributes within mutually dependent fields by incorporating data-driven heuristics rather than relying on left-most selection, 2) extending normalization capabilities to Boyce–Codd Normal Form and higher, 3) and developing mechanisms to distinguish true functional dependencies from those that arise due to limited or biased samples. Additional enhancements include supporting noisy datasets through robust handling of missing or erroneous values and providing connectors for common relational database systems or ETL tools, which would allow the library to function as a drop-in module within modern data engineering workflows and increase adoption.

REFERENCES

[1] A. Mitrovic. “A Web-Enabled Tutor for Database Normalization”. In Proceedings of the International Conference on Computers in Education (ICCE), pp. 1276-1280, December 2002.

[2] D. Mendoza and N. Piedra, “TutNorBD: Assistant for teaching and learning process of relational database normalization up to 3NF from a universal table”. In 2020 XV Conferencia Latinoamericana de Tecnologias de Aprendizaje (LACLO) (pp. 1-8). IEEE, October 2002.

[3] N. Mendjoge, A. R. Joshi, and M. Narvekar, “Intelligent tutoring system for Database Normalization”. In 2016 international conference on computing communication control and automation (ICCUBEA) (pp. 1-6). IEEE, August 2016.

[4] H. I. Piza, L. F. Gutiérrez, and V. H. Ortega, “An educational software for teaching database normalization”. Computer Applications in Engineering Education, 25(5), 812-822, June 2017.

[5] H. Du, and L. Wery, “Micro: A normalization tool for relational database designers”. Journal of Network and Computer applications, 22(4), 215-232, October 1999.

[6] A. Yazici, and Z. Karakaya, “JMathNorm: A database normalization tool using mathematica”. In International Conference on Computational Science (pp. 186-193). Berlin, Heidelberg: Springer Berlin Heidelberg, May 2007.

[7] P. B. Alappanavar, R. Grover, S. Hunjan, D. Patil, and Y. Girnar. “Automating the normalization process for relational database model”. International Journal of Engineering Research and Applications (IJERA), 3(1), 1826-1831, January 2013.

[8] R. Vangipuram, R. Velputa, and V. Sravya, “A Web Based Relational database design Tool to Perform Normalization”. International Journal of Wisdom Based Computing, 1(3), 15-23, December 2011.

[9] M. Demba (2013). Algorithm for relational database normalization up to 3NF. International Journal of Database Management Systems, 5(3), 39.

[10] Y. Dongare, P. Dhabe, and S. Deshmukh, “RDBNorma: A semi-automated tool for relational database schema normalization up to third normal form”. International Journal of Database Management Systems, Vol. 3, No. 1, March 2011.

[11] E. Akadal, and M. H. Satman, "A Novel Automatic Relational Database Normalization Method”. Acta Informatica Pragensia, 11(3), 293-308, December 2022.

[12] IMDb Non-Commercial Datasets. IMDb, March, 2024. https://developer.imdb.com/non-commercial-datasets/

[13] T. Connolly, and C. Begg, “Database systems: A practical approach to design, implementation, and management. 6th Edition”, 2021.