# A Prototypical Tool for Analyzing Functional Dependencies Induced from Spreadsheets

Sergio Alejandro Gómez[1,2] and Pablo Rubén Fillottrani[1,2]

[1] Laboratorio de I+D en Ingeniería de Software y Sistemas de Información (LISSI)
Departamento de Ciencias e Ingeniería en Computación
Universidad Nacional del Sur
San Andrés 800, (8000) Bahía Blanca, Argentina
{sag,prf}@cs.uns.edu.ar
[2] Comisión de Investigaciones Científicas de la Provincia de Buenos Aires

**Abstract.** We present an extension to the GF framework for OntologyBased Data Access with the aim of determining the functional dependencies that hold in a spreadsheet. Spreadsheets are restricted to a single table expressed as a CSV text file. An initial set of tentative functional dependencies is computed using the TANE datamining algorithm. This set is then presented to the user who is used as an oracle to revise it. Given a functional dependency, the user can see the tuples from the spreadsheet justifying it. The user can revise the validity of the functional dependency with the help of our system, which will generate tuples not present in the dataset by using values already present in the table. The user can then add some of the new records to the table when he considers their feasibility and rerun the miner to see if the functional dependency still holds. We present a running example along with a downloadable JAVA-based application with source code of the miner in the C programming language and the files used in our experiments to help with the reproducibility of our results.

**Keywords.** Spreadsheets, TANE, Functional dependencies, Databases.

## 1. Introduction

Spreadsheets are an essential tool for organizations as they provide a simple and flexible way to store, analyze, and visualize data. They allow for efficient data management, calculations and data analysis, enabling informed decision-making. They also support collaboration, allowing multiple users to work together on the same data set. Data in spreadsheet tables can sometimes face challenges in terms of organization due to several reasons. Firstly, spreadsheet tables lack the structure and strict constraints of a database, making it easier for inconsistencies and errors to occur. Without predefined data types and constraints, it becomes more challenging to ensure data integrity and enforce data organization rules.

Thus, while spreadsheets offer convenience and flexibility, they are not inherently optimized for data organization. To overcome these challenges, organizations often rely on more robust data management systems, such as databases, that provide structured schemas, data validation, and stronger organization capabilities. Normalized databases play a crucial role in data management by reducing redundancy and improving data integrity. The discovery of functional dependencies is an essential step in the normalization process. Functional dependencies are relationships between attributes in

a database. They identify the dependencies between the values of one set of attributes on another set of attributes. By discovering and analyzing these dependencies, we can identify the key determinants in a dataset and eliminate data redundancy. Normalization helps in achieving a more efficient and organized database structure. It eliminates data anomalies such as update, insertion, and deletion anomalies that can occur due to redundant or inconsistent data. Thus, the discovery of functional dependencies plays a crucial role in achieving these benefits and establishing a well-structured and optimized database design.

Algorithms for the discovery of functional dependencies, such as TANE (Topological Attribute Noise Elimination) [1], are essential tools in data management and database design. They play a crucial role in identifying and understanding the relationships between attributes in a dataset, enabling data cleaning and normalization processes and assisting in the identification of candidate keys, which are essential for designing well-structured relational databases.

In this paper, our goal is to tackle the problem of finding functional dependencies in a spreadsheet table. In brief, our semi-automatic proposal consists of: first, the spreadsheet table is represented as single datasource $D$ comprised of a CSV plain text file with fields separated by either commas or semicolons; second, we run the TANE datamining algorithm over $D$ to obtain a set $S$ of candidate functional dependencies supported by the data; third, the set $S$ is presented to the user who will act as an oracle in determining the viability of each candidate functional dependency in $S$. For this, the user can select a particular functional dependency $f$ and the system shows the projection $P$ of the current tuples (restricted to the fields referenced in $f$) in $D$ supporting the functional dependency and also generates a new set $N$ of potential tuples that are not currently present in $D$. The user can select a subset $S_N$ of $N$ containing some of these potential tuples add them to the tuples already present in $D$, building in fact a new set $P \cup S_N$. The TANE algorithm is then executed against $P \cup S_N$ for determining if $f$ is still valid and what new functional dependencies $N_f$ are discovered. The user has now the chance of determining if he wants to delete $f$ from $S$ and also if he wants to add some of the elements of $N_f$ to the set $S$ of candidate functional dependencies. Thus, the main contribution of this work is (i) proposing a method for determining a set of functional dependencies from a single spreadsheet table; (ii) provinding a functional prototypical implementation of the proposed approach, comprised of JAVA stand-alone program integrated into our GF framework for ontology based- data access; (iii) a set of examples to show how the approach works, and (iv) a modification of a third-party implementation of the TANE algorithm for being used independently from our system. We include source code and a functional executable file published online along with the data files to reproduce the results presented here. A running example is provided to illustrate our approach.

The rest of the paper is structured as follows. In Sect. 2, we briefly revisit the notion of CSV file. In Sect. 3, we discuss how to find functional dependencies in CSV files and the modifications we made to a third-party data miner published online. In Sect. 4, we present the module added to GF to revise functional dependencies in the CSV file. In Sect. 5, we review related work. In Sect. 6, we conclude and foresee future work.

## 2. Spreadsheets as CSV Files

Excel is a popular spreadsheet software developed by Microsoft. They consist of multiple worksheets, each containing a grid of cells organized into rows and columns. These cells can hold various types of data, such as numbers, text, formulas, and functions. Excel spreadsheets offer a wide range of features and functionalities for data manipulation, analysis, and visualization, including formatting options, sorting and filtering capabilities, and chart creation. In Fig. (1.a), we show a simple Excel table for representing information of owners of cars along with information of the cars themselves. The information of the owners is comprised of the columns *IDPerson* and *Name*. Cars are represented by the columns *IDCar*, *CarBrand* and *CarPerception*. Notice that the column *CarPerception* represents the perception of the particular brand by a particular user.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | IDPerson | Name | IDCar | CarBrand | CarPerception |
| 2 | 1 | John | 1 | Aston Martin | high |
| 3 | 2 | Mary | 2 | Fiat | low |
| 4 | 3 | Paul | 3 | Audi | medium |
| 5 | 1 | John | 4 | BMW | medium |
| 6 | 1 | John | 5 | Chevrolet | low |

(a)

```
IDPerson;Name;IDCar;CarBrand;CarPerception
1;John;1;Aston Martin;high
2;Mary;2;Fiat;low
3;Paul;3;Audi;medium
1;John;4;BMW;medium
1;John;5;Chevrolet;low
```

(b)

**Fig.1.** (a) Spreadsheet for owners of cars and (b) CSV code of the spreadsheet

CSV files are a simpler and more universally compatible format for tabular data. CSV files are plain text files that store tabular data as a series of values, with each value separated by a comma. Each line in the file represents a row of data, and the comma acts as a delimiter to separate individual values within each row. CSV files do not support formatting, formulas, or multiple worksheets like Excel spreadsheets. However, they are lightweight, easy to read and write, and can be opened by various software applications, making them widely used for data exchange between different programs and systems. For instance, in Fig. (1.b), we present the CSV version of the Excel table of Fig. (1.a). Notice that in this particular case, the field separator is the semicolon character.

In previous publications ([2] and references there in), we have been reporting about the development of a framework for Ontology-Based Data Access called GF. In this work, we extend such framework with the aim of solving the problem of finding functional dependencies hidden in CSV tabular data. The proposal presented in this work has been integrated with such application. A closedsource stand-alone JAVA application is available online[1] to reproduce the results presented here. Also an open source extension of the TANE miner described in the next section is available for downloading and use.

## 3. Finding Functional Dependencies in Spreadsheets

[1] See http://cs.uns.edu.ar/~sag/cacic2023

In the context of relational databases, a functional dependency is a relationship between two sets of attributes within a database table, that describes the dependence of one set of attributes (known as the dependent attributes) on another set of attributes (known as the determinant attributes). That is, given the values of the determinant attributes, the values of the dependent attributes can be determined. Functional dependencies play a crucial role in database design and normalization, helping to ensure data integrity and minimize data redundancy.

TANE (Topological Attribute Noise Elimination) [1] is an algorithm used for mining functional dependencies from a given relational database table. It is based on the concept of the topological sort and utilizes pruning techniques to efficiently discover all the non-redundant functional dependencies within the table. A brief explanation of the TANE algorithm is as follows:

1. *Input:* The algorithm takes a relational database table as input, consisting of a set of attributes and their corresponding values.
2. *Candidate Generation:* Initially, TANE starts with a set of candidate functional dependencies that include individual attributes and pairs of attributes. For example, if the table has attributes $A$, $B$, and $C$, the initial set candidates would be $\{A \rightarrow B, A \rightarrow C, B \rightarrow A, B \rightarrow C, C \rightarrow A, C \rightarrow B\}$.
3. *Pruning:* The algorithm employs pruning techniques to eliminate redundant candidates. It checks if each candidate can be further extended by adding more attributes without violating the closure property. If a candidate is found to be redundant, it is removed from the set of candidates.
4. *Topological Sorting:* TANE performs a topological sorting of the attributes based on their dependencies to ensure that dependencies are discovered in a particular order. It determines the dependencies between attributes by computing the closures of attribute sets. The closure of an attribute set is the set of all attributes that can be determined based on the given set of attributes.
5. *Dependency Discovery:* TANE iterates through the topologically sorted attribute order and discovers functional dependencies by checking if each candidate is satisfied by the current set of attributes. If a candidate is satisfied, it is considered a valid functional dependency and added to the result set.
6. *Closure Pruning:* After discovering each dependency, TANE applies closure pruning to eliminate any remaining redundant candidates that are no longer necessary based on the dependencies found so far.
7. *Repeat Step*: Steps 4 to 6 are repeated until no more dependencies can be discovered.
8. *Output:* The final result set contains all the non-redundant functional dependencies that have been mined from the input table.

Our approach to finding the functional dependencies holding in a spreadsheet starts by running a TANE miner on the CSV file contents. For doing this, we adapted an already existing TANE implementation.[2] That implementation is a console application based on the C programming language. It takes as input a CSV file and as output it

[2] See https://github.com/getterk96/Database-Functional-Dependency-Digging-Algorithms.

prints to the screen the functional dependencies that it has found. The dependencies found are listed as field indexes.

From that starting point, we extended that miner for having a more friendly interface that allows to use it as parameterized command-line application where the input file, output file, temporary file, and field-separator character can be specified by a prospector user. In Fig. 2, we can see an example of how to use the improved utility on the data presented in Fig. (1.b). The input file named Owner-Car.csv is preceded by the *-i* switch. Notice that the temporary file (preceded by the *-t* switch) is needed to redirect the original output of the miner and the output file (preceded by the *-o* switch) is only useful to a client/user. Additionally, the *-s* switch allows to specify a separator character for fields (viz., either a semicolon or a comma).

The functional dependencies found with the miner are presented in Fig. 3. We call these dependencies, *tentative functional dependencies*. Notice that some of these functional dependencies are the true ones (e.g. *IDPerson* → *Name*) but other ones are just contingent on the values present in the data (e.g. *Name* → *IDPerson*). The revision of these dependencies to discard the false ones from the set of tentative functional dependencies is the subject that the module that we have added to GF deals with, and that is the matter of the next section.

```
TANE-Sergio.exe -i “Owner-Car.csv” -o “result-owner-car.txt” -t
                “temporal-owner-car.txt” -s “;”
```

**Fig.2.** Improved interface for the TANE miner

```
             IDPerson →  Name                IDCar → Name
IDPersonCarPerception →  IDCar               IDCar → CarBrand
IDPersonCarPerception →  CarBrand            IDCar → CarPerception
                 Name →  IDPerson         CarBrand → IDPerson
    NameCarPerception →  IDCar            CarBrand → Name
    NameCarPerception →  CarBrand         CarBrand → IDCar
                IDCar →  IDPerson         CarBrand → CarPerception
```

**Fig.3.** Functional dependencies found by the TANE miner in the *CarOwner* spreadsheet

## 4. Revising Functional Dependencies

Here, we present the new module in GF that allows a user to interactively revise functional dependencies computed from a CSV datasource. In Fig. 4, we present the user interface for using the new module.

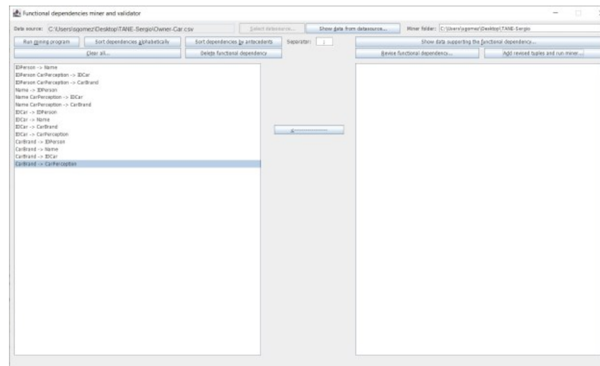

**Fig.4.** Module for mining and validating functional dependencies in CSV files

The controls in the GUI are enabled in a strict order that allows the user to unlock functionalities when input data is available for using them. This order is specified by the state diagram in Fig. 5. The usual workflow a user would follow when using the application will be: (1) opening the form takes him to state $q_0$; (2) specifying the datasource by selecting the particular CSV file he wants to work with (thus going into state $q_1$); (3) at any time the data from the data source can be explored; (4) running the dataminer as the one described in Sect. 3 computes the functional dependencies that are shown on the list on the left (notice that the miner program can be changed for an alternative program provided that it satisfies the interface as explained in Fig. 2), this will take the user to state $q_2$; (5) once the list is filled with the functional dependencies computed from the current data in the CSV, which we chose to call *tentative functional dependencies*, as they are contingent on current data, the user can select one of them getting to state $q_3$; (6) once in $q_3$, the user can optionally chose directly delete a tentative functional dependency or else to see the projection of the table limited to the columns mentioned in the selected functional dependency to determine if some combination of values is missing, determining so by pressing the button labeled “Revise FD” will take him to state $q_4$; (7) in $q_4$ the application will open a new form showing to the user invented tuples that are not currently in the table by producing random combinations with data already present in the table according to the algorithm in Fig. 7; (8) in $q_5$, the user can select one or more than one alternative tuples to add to the already existing tuples in the projected table, and then go to $q_6$; (9) in $q_6$, the miner is run on the projected table to validate if the revised dependency still holds or new ones appear, this new computed dependencies are displayed on the list located on the right, leading the system to state $q_7$; (10) in $q_7$, the user can delete the revised funtional dependency or select some of the newly found functional dependencies and copy them to the list on the left pane, and, (11) finally, the user can clear the process to return to state $q_0$.

Regarding step (2), our approach assumes that all fields are considered string typed, that no field delimiters are present and the user can choose only between comma and semicolon characters as field separators. Step (4) was already discussed in Sect. 3. To illustrate how steps (5)–(10) are intended to be used, we continue the preceding example. Consider the tentative functional dependency *CarBrand* → *CarPerception* computed in (4). The projection of the tuples restricted to its fields, and without duplicate data, is shown in Fig. (6.a). For example, suppose that the user considers that

the perception of *Chevrolet* must be *high* instead of *low*. The user can ask the system to generate unseen tuples in steps (6)-(7). This new invented tuples are generated using the algorithm in Fig. 7. Once the user explores the invented tuples, he can select the appropriate ones to be added to the projection (see Fig. 6.b). Then, after running again the miner, in this particular case, there are no functional dependencies that hold in this view of the table with the invented record that were selected. So the user can decide to delete the functional dependency under consideration. Conversely, as the system generates those potential scenarios, it lets the user deciding to keep or discard the proposed functional dependency. When the user observes that a suggested combination of data from the domain that respects the functional dependency is not consistent with his view of the domain, he can opt for discarding the suggested dependency. Besides, if other functional dependencies were discovered to hold in this new dataset, they will be shown in the list of right side of the screen, and then the user can decide to add to the main set of tentative functional dependencies.

## 5. Discussion and Related Work

Cunha et al. [3] present techniques and tools to transform spreadsheets into relational databases and back. A set of data refinement rules is introduced to map a tabular datatype into a relational database schema. Having expressed the transformation of the two data models as data refinements, they obtain for free the functions that migrate the data usin well-known relational database techniques to optimize and query the data. Because data refinements define bi-directional transformations, they can map such database back to an optimized spreadsheet. They implemented the data refinement rules and constructed Haskell-based tools to manipulate, optimize and refactor Excel-like spreadsheets. In our prototypical approach deals with Excel files but expressed as CSV tables, GF has the functionality to load Excel files and this feature can be added in the future. In our application, CSV files are handled internally as H2 tables but the data miner that we use as an external tool handles it as plain text. The work of Cunha et al. uses the set of functional dependencies to generate a normalized relational

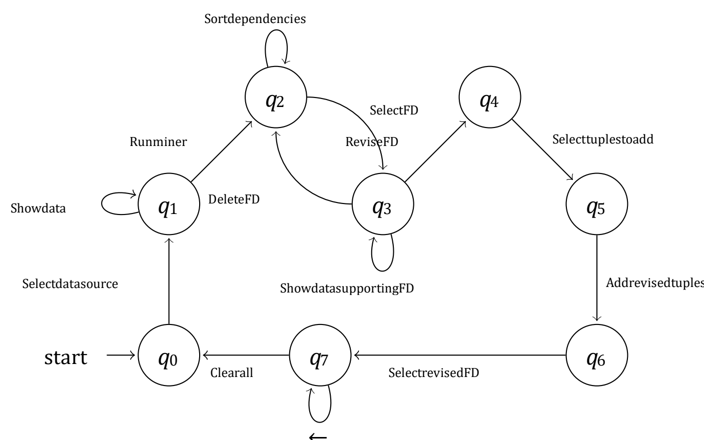


**Fig.5.** State diagram describing the usage of the GUI controls

Records justif...

| CarBrand | CarPerception |
|---|---|
| Aston Martin | high |
| Audi | medium |
| BMW | medium |
| Chevrolet | low |
| Fiat | low |

(a)

Revise functional dependency with potential values: ...

| CarBrand | CarPerception |
|---|---|
| Aston Martin | low |
| Fiat | high |
| Audi | high |
| Fiat | medium |
| BMW | high |
| Audi | low |
| Chevrolet | high |
| BMW | low |
| Chevrolet | medium |
| Aston Martin | medium |

(b)

**Fig.6.** (a) Records justifying functional dependency *CarBrand* → *CarPerception* and (b) Invented records with alternative values

```
Algorithm GenerateAlternatives(Table T, FunctionalDependency C1...Cn → D)
Let values be a (n + 1)-size vector of sets
for i := 1 to n do
   values_i := executeQuery(SELECT DISTINCT C_i FROM T)
values_n+1 := executeQuery(SELECT DISTINCT D FROM T)
Let script be an empty sequence of strings
script.addLine(CREATE TABLE aux(C1 VARCHAR(100), ..., Cn VARCHAR(100), D
VARCHAR(100));)
for epoch := 1 to MAX TUPLES do
   Let x be a vector of n + 1 components
   for i := 1 to n + 1 do x_i := randomElementFrom(values_i)
   if x is a previously unseen combination of values then
      result := executeQuery(SELECT COUNT (*) AS Result FROM T
         WHERE C1 = x1 AND ...AND Cn = xn AND D = x_n+1)
      if result = 0 then
         script.addLine(INSERT INTO aux(C1, ..., Cn, D) VALUES (x1, ...,
         x_n+1);)
ExecuteSQLScriptToGenerateTableAndShow(script)
```

**Fig.7.** Algorithm for generating unknown alternatives from table *T* w.r.t. functional dependency $C_1 \ldots C_n \rightarrow D$

database from the contents of the spreadsheet. Our application does not do that yet and adding that functionality remains a future work. They use the FUN mining technique and we use the TANE mining technique. As our approach is fully customizable, we could employ the FUN miner in the future provided a functioning version of it is available. In particular, Cunha et al. present an example that considers a spreadsheet for representing a property renting system (see [3, Fig. 1]). In the accompanying files published online with our executable application, we provide a recreation of that file, named book.csv, showing that our system can compute the functional dependencies shown in that work.

Despite the ubiquity of spreadsheets, the problem of dealing with transformations of spreadsheet data to more formal data formats is still relevant. Müller and Mertov´a [4] justify and propose a lightweight recording-based solution to the tracing of the steps for transforming spreadsheets into ontologies that works on a wide variety of spreadsheet programs, from Microsoft Excel to Google Docs. GF can transform spreadsheets into ontologies but in this particular work we are concentrated in functional dependencies emerging from tables within spreadsheet data.

Salem and Abdo [5] propose two techniques for mining accurate conditional functional dependencies rules from such databases to be employed for data cleaning. The idea of the proposed techniques is to mine firstly maximal closed frequent patterns, then mine the dependable conditional functional dependencies rules with the help of lift measure. That approach is complementary to ours because in the current status of our work, we assume that the data is accurate. Their approach could be used to extend our solution with a data-cleaning preprocess for capturing semantic errors.

## 6. Conclusions and Future Work

In this paper we tackled the problem of finding functional dependencies in a spreadsheet table. We relied on the TANE datamining algorithm integrating it with our GF framework for ontology-based data access. We presented a running example that showed how our approach assists the user so he can revise a tentative set of functional dependencies computed with TANE using possible tuples not already present in the table that are computed by our system. The limitations of our approach are all fields are considered string typed, that no field delimiters are present and the used can choose only between comma and semicolon characters as field separators. The solution is customizable provided that a miner is provided satisfying a very precise command-line interface.

Possible research avenues for extending the work presented here include generating a normalized data base using the functional dependencies computed by the miner and later revised by the user with the assistance of our system, and then exporting this database as an ontology. Other possible paths to be explored are the adaptation of the mining algorithms to different data types and constraints.

*Acknowledgments.* This work was supported by Secretaría General de Ciencia y Técnica, Universidad Nacional del Sur, Argentina, and by Comisión de Investigaciones Científicas de la Provincia de Buenos Aires (CIC-PBA).

## References

1. Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.: TANE–An efficient algorithm for discovering functional and approximate dependencies. The Computer Journal **42**(2) (1999) 100–111
2. Gómez, S.A., Fillottrani, P.R.: A Query-By-Example Approach to Compose SPARQL Queries in the GF Framework for Ontology-Based Data cmdAccess. In Pesado, P., ed.: 28th Argentine Congress, CACIC 2022 – Revised Selected Papers. Springer (2023) 211–226
3. Cunha, J., Saraiva, J., Visser, J.: From spreadsheets to relational databases and back. In: PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation. ACM (jan 2009) 179–188
4. Müller, W., Mertová, L.: ReStoRunT: Simple Recording, Storing, Running and Tracing changes in Spreadsheets. In et al., B.K.R., ed.: BTW 2023, Lecture Notes in Informatics (LNI), Gesellschaft für Informatik. LNI (2023) 865–877
5. Salem, R., Abdo, A.: Fixing rules for data cleaning based on conditional functional dependency. Future Computing and Informatics Journal **1**(1) (2016) 10–26