A Method for Normalization of Relation Schema Based on Data to Abide by the Third Normal Form

HYONTAI SUG
Department of Computer Engineering
Dongseo University
47 Jurye-ro, Sasang-gu, Busan 47011
REPUBRIC OF KOREA

Abstract: - Databases play an important role in applied mathematics, and normalization for relational databases is very important to avoid anomalies of relations which may not be in normalized forms of the third normal forms. But, normalization may be a difficult task, since the designers of the databases may not fully understand the domain of each attribute that are contained in the relation schema or they may not have full understanding about the concept of normalization. In this paper an efficient method that checks the possibility of the need of further normalization using stored data in relations is presented based on possible functional dependencies between attributes in the relations. By checking possible functional dependencies, the database designers can determine the need of further normalization, and may improve the structure of the relation schemas. Experiments were performed for an example of relational database that can be found in the organization of tutorial of MySQL which is a representational database management system, and the experiments showed good results.

Key-Words: - Applied Mathematics, Algorithms, Information Theory, Normalization, Relational Databases, Normal Forms, Functional Dependency, Database Systems

Received: December 10, 2019. Revised: May 5, 2020. Accepted: May 19, 2020. Published: May 27, 2020.

1 Introduction

Relational databases are very important ingredients for modern information society. Nowadays, most operational databases are based on relational databases [1]. An operational database creates and updates large amounts of data in real time, and transaction processing is a key technology to support concurrency, integrity, and recoverability [2]. Data integrity is very important for operational databases of enterprises, because incorrect data will generate incorrect outputs, as a results, the incorrect outputs may lead to bad or wrong decisions. Duplicate data in relations may cause inconsistent data if we miss updating anyone of them, as the data are updated with time. The inconsistent data may cause to generate incorrect information. In order to avoid data inconsistency and other anomalies, it is recommended that a relation schema should be at least in the third form.

Designing the structures of relation schemas requires allocating appropriate attributes for each relation schema, and requires checking functional dependencies between the attributes in the relational schema for possible future data inconsistency and anomalies. The checking process is usually done manually so that there is always some possibility that there are unnoticed functional dependencies between non-key attributes or functional dependency of non-key attributes on a part of the primary key when the key is a composite key. Inappropriate functional dependencies make the relational schemas not in the third or second normal forms, and the design mistakes could make the related relation schemas have inconsistent data, and could generate anomalies as the data are updated, inserted, and deleted with time. But, it may not be easy to detect such design mistakes, because humanities are not accustomed to recognize their own mistakes.

A lot of research has been done to discover functional dependencies efficiently for large data sets in table form as an optimization problem of time complexity, because we can have 2^m combinations of attributes for a table having m attributes. The algorithms for discovering functional dependencies can be categorized into three or four approaches; top-down, bottom-up, and hybrid approach, and some others [3, 4]. Top-down approach like TANE [5] or TANE-based incremental algorithm [6] generate the lattice of attributes first to generate candidate functional dependencies, and the candidate functional dependencies are tested for validity using real data. Bottom-up approach algorithm like FastFD generates so called difference sets and agree sets of attributes based on some tuple pairs in the table [7].

The sets are used to drive all functional dependencies. Hybrid approach algorithm mixes the good points of the two approaches, and reports better performance [4]. FDEP generates negative cover and positive cover based on FD-tree by pair-wise comparison of all tuple pairs in a table [8]. In [9] sampling-based algorithm is suggested to find approximate functional dependencies. The time complexity of the algorithms is polynomial times because the algorithms except the sampling based algorithm check the validity of candidate functional dependencies repeatedly based on data. Moreover, most of the experiments for the algorithms are based on data sets not exactly belonging to relations like the data sets in UCI machine learning repository [10]. The structure of data sets used for experiments is similar to views in relational databases. Note that views are usually made by joining several relations together so that the possibility of functional dependencies in the data sets could be increased.

Therefore, in this paper we want to check possible functional dependencies for relations efficiently based on data in relations, and want to check the utility of our suggested method in improving the structure of relation schemas for them to be in the third form.

2 Problem Formulation

The constraints for normal forms are well described in the textbooks for databases [2, 11]. We should check the functional dependencies of each relation schema or relation variable to confirm that the relation schema is at least in the third form. In order to check functional dependencies between attributes in a relation, we need to utilize Armstrong's axioms of inference rules.

2.1 Armstrong's axioms

Armstrong's axioms have three inference rules called reflexivity, augmentation, and transitivity.

Let R be a relation schema over the set of attributes U, and X, Y, Z be any subset of U.

1) Reflexivity: if $Y \subseteq X$, then $X \rightarrow Y$.
2) Augmentation: if $X \rightarrow Y$, then $XZ \rightarrow YZ$.
3) Transitivity: if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

We can use the above three inference rules to generate the closure of given set of functional dependencies of a relation schema.

An additional inference rule for our task of functional dependency checking is that if the left hand sides of two functional dependencies are the same, the two functional dependencies can be combined together with the same left hand side. That is, if we have $X \rightarrow Y$ and $X \rightarrow Z$, we have $X \rightarrow YZ$. Because of $X \rightarrow Z$, we have $X \rightarrow XZ$ by augmentation, and because of $X \rightarrow Y$, we have $XZ \rightarrow YZ$ by augmentation. Therefore, we have $X \rightarrow YZ$ by transitivity.

But, we don't have to combine two functional dependencies together, if the right hand sides of two functional dependencies are the same, even though the two functional dependencies can be combined together with the same right hand side. That is, if we have $X \rightarrow Z$ and $Y \rightarrow Z$, we have $XY \rightarrow Z$. Because of $Y \rightarrow Z$, we have $XY \rightarrow XZ$ by augmentation, and because of $X \rightarrow Z$, we have $XZ \rightarrow Z$ by augmentation. Therefore, we have $XY \rightarrow Z$ by transitivity. But, because each left hand side alone can functionally determine the same right hand side, we don't need to combine two functional dependencies together for simplicity. Note also that if $XY \rightarrow Z$, we don't have $X \rightarrow Z$ and $Y \rightarrow Z$ automatically.

2.2 Suggested method

We should check whether a given relation schema is in the third normal form. In order to check it there are two cases to check. In case of the primary key is a composite key, functional dependency from part of the composite key to non-key attributes has to be checked. After checking it functional dependency between non-key attributes has to be checked to make it sure that the given relation schema is in the third normal form. In case of the primary key is not composite key, checking functional dependency between non-key attributes is good enough for the task. We check functional dependency between single attributes first. Then, in case we find the same left hand side in the found functional dependencies, we combine them together as explained in section 2.1. In addition, checking on non-key multiple attributes as left hand side of a possible functional dependency will be performed when the attributes have close relationship only, because such case is rare in a relation schema.

2.2.1 Case 1: the primary key is single attribute

When the primary key consists of single attribute, we should check many to one or one to one correspondence between chosen non-key attributes to make it sure that it is in the third normal form. We may or may not skip a non-key attribute that has key-like characteristics, like name attribute, to avoid unnecessary calculation.

PROCEDURE 1:
INPUT: a relation **r**, chosen set of attributes

OUTPUT: many to one correspondence between attributes
BEGIN
1. Check many to one correspondence for each pair of input attributes in **r** in both directions;
2. Output possible functional dependencies if found;
3. Combine found possible functional dependencies together, if they have the same right had side;

END.

Note that we have ${}_nC_2$ combinations of attribute pairs for the task numbered 1 in the above procedure. Checking possible functional dependency in both directions means that we should check $X \rightarrow Y$ as well as $Y \rightarrow X$ based on stored data in **r**. We may use a sorting based algorithm or one of the algorithms introduced in section 1 for this task.

2.2.2 Case 2: the primary key is composite key

When the primary key consists of multiple attributes, we should check many to one or one to one correspondence between each part of the composite key and non-key attributes to make it sure that it is in the second normal form. After that we can perform procedure 1 to check whether the given relation schema is in the third normal form or not.

PROCEDURE 2:
INPUT: a relation **r** having composite key
OUTPUT: many to one correspondence between attributes
BEGIN
1. Check many to one correspondence for each pair consisting of all parts of composite key and non-key attributes in **r** in one direction of being from the parts of composite key to non-key attributes;
2. Output possible functional dependencies if found;
3. Perform procedure 1;

END.

We only have to do checking on possible functional dependency for the parts of composite key and non-key attributes in one direction to regulate the constraints of the second normal form.

3 Problem Solution

MySQL sample database was used to illustrate the checking process. Experiments for all the relations were performed using an example database provided by the organization of MySQL tutorial. The sample database can be downloaded from MySQL tutorial site [12]. MySQL database management system is especially very popular in databases for websites. According to Datanyze, it is the number one database management system in use cases [13] and freely available. The sample database has eight relations; productlines, employees, offices, products, customers, orderdetails, orders, payments. The primary key is underlined in each relation schema or relation variable in the followings. Relation schema or relation variable means the heading part of a relation where a relation consists of heading and body part. The schema of each relation is as follows.

Customers(customerNumber, customerName, contactLastName, contactFirstName, phone, addressLine1, addressLine2, city, state, postalCode, country, salesRepEmployeeNumber, creditLimit)
Employees(employeeNumber, lastName, firstName, extension, email, officeCode, reportsTo, jobTitle)
Offices(officeCode, city, phone, addressLine1, addressLine2, state, country, postalCode, territory)
Orders(orderNumber, orderDate, requiredDate, shippedDate, status, comments, customerNumber)
Orderdetails(orderNumber, productCode, quantityOrdered, priceEach, orderLineNumber)
Payments(customerNumber, checkNumber, paymentDate, amount)
Products(productCode, productName, productLine, productScale, productVendor, productDescription, quantityInStock, buyPrice, MSRP)
Productlines(productLine, textDescription, htmlDescription, image)

The functional dependency between non-key attributes were checked for each relation, and many possible functional dependencies were found based on stored data in each relation. In addition, because orderdetails and payments relation schema have composite key, they were checked additionally whether they are in the second normal form or not.

3.1 Customers relation

Customers relation has 122 tuples. We check possible functional dependencies for customers relation for each combination of non-key attributes. We check the attribute pair customerName and all the other non-key attributes first in the set of non-key attributes, {customerName, contactLastName, contactFirstName, phone, addressLine1, addressLine2, city, state, postalCode, country, salesRepEmployeeNumber, creditLimit}.

3.1.1 Checking on customerName attribute first

The found possible functional dependencies between each pair of non-key attributes when we consider attribute customerName first in relation customers are as follows.

customerName → contactLastName

customerName → contactFirstName

customerName → phone

phone → customerName

customerName → addressLine1

addressLine1 → customerName

customerName → addressLine2

customerName → city

customerName → state

customerName → postalCode

customerName → country

customerName → salesRepEmployeeNumber

customerName → creditLimit

In short, we found a possible functional dependency,

customerName → {contactLastName, contactFirstName, phone, addressLine1, addressLine2, city, state, postalCode, country, salesRepEmployeeNumber, creditLimit}, and two other possible functional dependencies,

phone → customerName, and

addressLine1 → customerName.

Note that non-key attributes of relation schema customers are customerName, contactLastName, contactFirstName, phone, addressLine1, addressLine2, city, state, postalCode, country, salesRepEmployeeNumber, creditLimit, so that attribute customerName functionally determines all the other non-key attributes when we check it based on stored values in the relation. We may leave this functional dependency untouched because we may have the same customer name which has different data in the rest of the non-key attributes. In other words, customerName attribute has key-like characteristics.

3.1.2 Checking on contactLastName attribute

The found possible functional dependencies between each pair of non-key attributes when we consider attribute contactLastName first in relation customers are as follows. Three possible functional dependencies were found.

phone → contactLastName

addressLine1 → contctLastName

contactLastName → addressLine2

Moreover, by transitivity in Armstrong's axioms, we have

phone → addressLine2, and

addressLine1 → addressLine2.

3.1.3 Checking on contactFirstName attribute

The found possible functional dependencies between each pair of non-key attributes when we consider attribute contactFirstName first in relation customers are as follows. Two possible functional dependencies were found.

phone → contactFirstName

addressLine1 → contctFirstName

3.1.4 Checking on {contactLastName, contactFirstName} attribute

Because a name consists of last name and first name, the two related attributes, contactLastName and contactFirstName can be considered as one attribute, so the same process was performed to the other non-key attributes to make pairs in checking possible functional dependencies. The followings are found possible dependencies.

{contactLastName, contactFirstName} →customerName
customerName → {contactLastName, contactFirstName}
{contactLastName, contactFirstName} → phone
phone → {contactLastName, contactFirstName}
{contactLastName, contactFirstName} → addressLine1
addressLine1 → {contactLastName, contactFirstName}
{contactLastName, contactFirstName} → addressLine2
{contactLastName, contactFirstName} → city
{contactLastName, contactFirstName} → state

{contactLastName, contactFirstName} → postalCode
{contactLastName, contactFirstName} → country
{contactLastName, contactFirstName} → salesRepEmployeeNumber
{contactLastName, contactFirstName} → creditLimit

In short, we found a possible functional dependency,

{contactLastName, contactFirstName} → {customerName, phone, addressLine1, addressLine2, city, state, postalCode, country, salesRepEmployeeNumber, creditLimit}, and three other possible functional dependencies,

cunstomerName → {contactLastName, contactFirstName}, and
phone → {contactLastName, contactFirstName}, and
addressLine1 → {contactLastName, contactFirstName}.

In short, attribute set {contactLastName, contactFirstName} has key-like characteristics, because they determine the values of all the other non-key attributes uniquely.

3.1.5 Checking on phone attribute

The found possible functional dependencies between each pair of non-key attributes when we consider attribute phone first in relation customers are as follows. Eight possible functional dependencies were found.

phone → addressLine1
addressLine1 → phone
phone → addressLine2
phone → city
phone → state
phone → postalCode
phone → country
phone → salesRepEmployeeNumber
phone → creditLimit

In short, we found a possible functional dependency,

phone → {addressLine1, addressLine2, city, state, postalCode, country, salesRepEmployeeNumber, creditLimit}, and one other possible functional dependency,

addressLine1 → phone.

3.1.6 Checking on addressLine1 attribute

The found possible functional dependencies between each pair of non-key attributes when we consider attribute addressLine1 first in relation customers are as follows. Seven possible functional dependencies were found.

addressLine1 → addressLine2
addressLine1 → city
addressLine1 → state
addressLine1 → postalCode
addressLine1 → country
addressLine1 → salesRepEmployeeNumber
addressLine1 → creditLimit

In short, we found a possible functional dependency,

addressLine1 → {addressLine2, city, state, postalCode, country, salesRepEmployeeNumber, creditLimit}.

3.1.7 Checking on addressLine2 attribute

No possible functional dependencies were found between each pair of non-key attributes when we consider attribute addressLine2 first in relation customer. Most rows of addressLine2 attribute have NULL values. The attribute has only 15 different values in 122 tuples like values in {1 Garden Road, 27-30 Merchant's Quay, 2nd Floor, 8 Temasek, 815 Pacific Hwy, Alessandro Volta 16, Bronz Apt. 3/6 Tesvikiye, Crowther Way 23, Floor No. 4, Level 11, Level 15, Level 2, Level2, NatWest Center #13-03, NULL}.

3.1.8 Checking on city attribute

The found possible functional dependency between each pair of non-key attributes when we consider attribute city first in relation customers is as follows.

city → country

The reason why this functional dependency exists in the relation is that the limitation of data, because different countries may have the same city name so that almost no functional dependency exists between city and country.

3.1.9 Checking on state attribute

No possible functional dependencies were found between each pair of non-key attributes when we

consider attribute state first in relation customer. Most rows of state attribute have NULL values.

3.1.10 Checking on postalCode, country, salesRepEmployeeNumber, creditLimit attribute

No possible functional dependencies were found between each pair of non-key attributes when we consider attribute postalCode, country, salesRepEmployeeNumber first in relation customer.

All in all, even though we have found many possible functional dependencies in customers relation, there is almost no room for further normalization, because each customer has unique non-key attribute values.

3.2 Employees relation

Employee relation has 23 tuples. So, the size of data is not good enough for possible functional dependency checking based on stored data in the relation. Anyway, we check possible functional dependencies for employees relation for each combination of non-key attributes. We check the attribute pair, extension and all the other non-key attributes first in the set of non-key attributes, {extension, email, officeCode, reportsTo, jobTitle}. We do not check possible functional dependencies of attribute set {lastName, firstName} because they have key-like characteristics.

3.2.1 Checking on extension attribute

The found possible functional dependencies between each pair of non-key attributes when we consider attribute extension first in relation employees are as follows.

email → extension

Because different email addresses can share the same extension number, there exists functional dependency, email → extension, of many to one.

extension → jobTitle

Because different extension numbers can share the same job title, for example, 'sales representative', there exists functional dependency of many to one of the above. By transitivity, we have additional functional dependency,

email → jobTitle

3.2.2 Checking on email attribute

The found possible functional dependency between each pair of non-key attributes when we consider attribute email first in relation employees is as follows.

email → officeCode

Because different email addresses can share the same office code, there exists functional dependency of the above, email → officeCode, of many to one.

3.2.3 Checking on officeCode, reportsTo, jobTitle attribute

No possible functional dependencies were found between each pair of non-key attributes when we consider attribute officeCode, reportsTo first in relation employees.

All in all, we have the following possible functional dependency in employees relation.

email → {extension, jobTitle, officeCode}.

Because we found a possible functional dependency, extension → jobTitle, further normalization may be considered.

3.3 Offices relation

Offices relation has eight non-key attributes of {city, phone, addressLine1, addressLine2, state, country, postalCode, territory}, but it has only seven tuples, so that checking possible functional dependencies for the relation for each combination of non-key attributes is meaningless. For example, we may have a possible functional dependency of many to one, city → {phone, addressLine1, addressLine2, state, country, postalCode, territory}, but it's meaningless because of not enough number of supporting tuples.

3.4 OrderDetails relation

OrderDetails relation has five attributes of { orderNumber, productCode, quantityOrdered, priceEach, orderLineNumber} and 2,996 tuples. Among them orderNumber and productCode make a composite key. Therefore, we have to check possible functional dependencies between orderNumber and non-key attributes as well as productCode and non-key attributes to

see that the relation schema satisfies the constraints of the second normal form. Moreover, possible functional dependencies between non-key attributes to see that the relation schema satisfies the third normal form.

3.4.1 Checking on orderNumber, productCode attribute

There is no functional dependency from orderNumber to non-key attributes, {quantityOrdered, priceEach, orderLineNumber} based on the stored values in the relation, and there is also no functional dependency from productCode to non-key attributes, {quantityOrdered, priceEach, orderLineNumber} based on the stored values in the relation, so that we can confirm that the relation schema is in the second normal form.

3.4.2 Checking on quantityOrdered, priceEach, orderLineNumber attribute

No possible functional dependencies were found between each pair of non-key attributes when we consider attribute QuantityOrdered, priceEach first in orderDetails relation.

3.5 Orders relation

Orders relation has 326 tuples. We check possible functional dependencies for orders relation for each combination of non-key attributes, {orderDate, requiredDate, shippedDate, status, comments, customerNumber} except the attribute comments, because the attribute can contain comments in natural language. We checked all the attribute pair of non-key attributes, and there are no possible functional dependencies found between non-key attributes based on the stored data.

3.6 Payments relation

Payments relation has four attributes of { customerNumber, checkNumber, paymentDate, amount} and 273 tuples. Among them customerNumber and checkNumber make a composite key. Therefore, we have to check possible functional dependencies between customerNumber and non-key attributes as well as checkNumber and non-key attributes to see that the relation schema satisfies the constrains of the second normal form. Moreover, possible functional dependencies between non-key attributes {paymentDate, amount} have to be checked to see that the relation schema satisfies the third normal form.

3.6.1 Checking on customerNumber attribute

There is no functional dependency from customerNumber to non-key attributes, {paymentDate, amount} based on the stored values in the relation.

3.6.2 Checking on checkNumber attribute

There are functional dependencies from checkNumber to non-key attributes, {paymentDate, amount} based on the stored values in the relation.

checkNumber → paymentDate

checkNumber → amount

amount → checkNumber

Because checkNumber attribute which is a part of the composite key functionally determines non-key attributes {paymentDate, amount} based on the stored values in the relation, further normalization may be considered. Or, we may think the reason why is that not enough number of tuples are stored in the relation, in other words, it's coincidence.

3.6.3 Checking on paymentDate, amount attribute

The found possible functional dependency between the non-key attributes based on stored data in the relation is as follows.

amount → paymentDate

But, attribute paymentDate does not functionally determine attribute amount based on the data. The reason for the above found possible functional dependency may be that not enough number of tuples are stored in the relation.

All in all, for payments relation we have an additional possible functional dependency by transitivity,

checkNumber → paymentDate

We have two other possible functional dependencies,

checkNumber →{ paymentDate, amount}

amount → {checkNumber, paymentDate}

3.7 ProductLines relation

The relation schema of productLines consists of attribute set, {productLine, textDescription, htmlDescription, image} where productLine is the primary key, and the relation has only seven tuples. The stored data for attributes htmlDescription and image are NULLs, and the stored values for attribute textDescription are explanation in natural language, so that we don't have to check possible functional dependencies for each combination of non-key attributes.

3.8 Products relation

We check possible functional dependencies for products relation for each combination of non-key attributes. The attribute productName is omitted for the checking because the attribute has key-like characteristics, and the attribute productDescription is omitted also, because the attribute has explanation in natural language. Products relation has 110 tuples. We check the attribute pairs, productLine and all the other non-key attributes first in the set of non-key attributes, {productLine, productScale, productVendor, quantityInStock, buyPrice, MSRP}.

3.8.1 Checking on productLine attribute

The found possible functional dependency between each pair of non-key attributes when we consider attribute productLine first in relation products is as follows.

quantityInStock → productLine

The above possible functional dependency has many to one relationship. For example, it has the mapping of attribute values between quantityInStock and productLine like {68, 1005, …} → {Classic Cars}, and there is no same amount of quantity in stock for different lines. But, we can infer that this phenomenon happened because of relatively small amount of tuples in the relation.

3.8.2 Checking on productScale attribute

The found possible functional dependency between each pair of non-key attributes when we consider attribute productScale first in relation products is as follows.

quantityInStock → productScale

The above possible functional dependency has many to one relationship. For example, it has the mapping of attribute values between quantityInStock and productScale like {68, 1049, …} → {1:12} and there is no quantity in stock for different product scales. But, we can infer that this phenomenon happened because of relatively small amount of tuples in the relation.

3.8.3 Checking on productVendor attribute

Two possible functional dependencies between each pair of non-key attributes were found when we consider attribute productVendor first in relation products. The first one is as follows.

quantityInStock → productVendor

The above possible functional dependency has many to one relationship. For example, it has the mapping of attribute values between quantityInStock and productVendor like {68, 600, …} → {AutoArt Studio Design}, and there is no same quantity in stock for different product vendors. But, we can inter that this phenomenon happened because of relatively small amount of tuples in the relation. The second one is as follows.

MSRP → productVendor

The above possible functional dependency has many to one relationship. For example, it has the mapping of attribute values between MSRP and productVendor like {60.67, 81.36, …} → {AutoArt Studio Design}, and there is no same MSRP for different product vendors. But, we can inter that this phenomenon happened because of relatively small amount of tuples in the relation.

3.8.4 Checking on quantityInStock attribute

Two possible functional dependencies was found between each pair of non-key attributes when we consider attribute QuantityInStock first in relation products. The first one is as follows.

quantityInStock → buyPrice

The above possible functional dependency has many to one relationship. For example, it has the mapping of attribute values between quantityInStock and buyPrice like {414, 540} → {33.3}, and there is no same quantity in stock for different product vendors. But, we can inter that this phenomenon happened because of relatively small amount of tuples in the relation.

The other possible functional dependency is as follows.

quantityInStock → MSRP

The above possible functional dependency has many to one relationship. For example, it has the mapping of attribute values between quantityInStock and MSRP like {6645, 8197} → {50.31}, and there is no same quantity in stock for different MSRPs. But, we can inter that this phenomenon happened because of relatively small amount of tuples in the relation.

3.8.5 Checking on buyPrice, MSRP attribute

No possible functional dependencies were found between non-key attributes when we consider attribute BuyPriceEach and MSRP in relation products.

All in all, we have the following possible functional dependencies for relation products.

quantityInStock → {productLine, productScale, productVendor, buyPrice, MSRP}, and
MSRP → productVendor

So, the attribute quantityInStock plays like a candidate key among non-key attributes, and MSRP functionally determines productVendor according to stored data. Therefore, further normalization may be considered.

4 Conclusion

Relational databases are important assets for online transaction processing in operational databases, for example, bank databases and airlines databases, and normalization is very important to avoid anomalies and data inconsistency in the relations of the databases. But, normalization may be a difficult task, since the designers of the databases may not fully understand the domain of each attribute that constitutes relation schemas, or they may not have full understanding about the concept of normalization. In this paper an efficient method that checks the possibility of the need of further normalization is suggested based on discovered possible functional dependencies between attributes in relations. The suggested method is examined using sample relations provided by the organization of MySQL tutorial, and showed the fact that some further normalization may be needed. By checking possible functional dependencies using the suggested method that investigates stored data in the relations efficiently, one may determine the need of further normalization as shown in the experiments.

References:

[1] D. Ramel, Database Trends Report: SQL Beats NoSQL, MySQL Most Popular, [https://adtmag.com/articles/2019/03/05/db-report.aspx], 2019.

[2] R. Elmasri and S.B. Navathe, *Fundamentals of Database Systems*, 7th ed., Pearson, 2017.

[3] J. Liu, J. Li, C. Liu, and Y. Chen, Discover dependencies from data – a review, *IEEE Transactions on Knowledge and Data Engineering*, Vol.24, No.2, 2012, pp. 251-264.

[4] T. Papenbrock and F. Naumann, A Hybrid Approach to Functional Dependency Discovery, *Proceedings of the 2016 International Conference on Management Data*, 2016, pp. 821-833.

[5] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen, TANE: An efficient algorithm for discovering functional and approximate dependencies, *The Computer Journal*, Vol.42, No.2, 1999, pp. 100-111.

[6] L. Caruccio, S. Cirillo, V. Deufemia, and G. Polese, Incremental Discovery of Functional Dependencies with a Bit-vector Algorithm, *Proceedings of the 27th Italian Symposium on Advanced database Systems*, 2019, pp. 146-157.

[7] C. Wyss, C. Giannella, and E. Robertson, FastFD: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances Extended Abstract, *Proceedings of the International Conference of Data Warehousing and Knowledge Discovery*, 2001, pp. 101-110.

[8] P.A. Flach and I. Savnik, Database dependency discovery: a machine learning approach, *AI Communications*, Vol.12, No.3, 1999, pp. 139-160.

[9] S. Kruse and F. Naumann, Efficient Discovery of Approximate Dependencies, *Proceedings of the VLDB Endowment,* Vol.11, No.7, 2018, pp 759-772.

[10] D. Dua and C. Graff, *UCI Machine Learning Repository* [http://archive.ics.uci.edu/ml] Irvine, CA, University of California, School of Information and Computer Science, 2019.

[11] C.J. Date, *Introduction to Database Systems,* 8th ed., Pearson, 2003.

[12] MySQLTUTORIAL.org, [https://www.mysqltutorial.org/mysql-sample-database.aspx], 2020.
[13] Datanyze, [https://www.datanyze.com/market-share/databases--272/mysql-market-share], 2020.