THIRD NORMAL FORM MADE EASY

Betty Salzberg
Northeastern University
Boston, Massachusetts

Third normal form is defined in different inequivalent ways in elementary data base textbooks. The method given in many of these books for decomposing a file into third normal form is non-algorithmic, depends on human recognition of a pattern, and has severe limitations and unexplained complexities. This is not necessary. There is an excellent algorithmic method available which is easy to learn, and which can be programmed.

The objective of this paper is to examine the differences concerning how third normal form is presented in various textbooks, and to present an easy way of introducing it to students.

This article appeared previously in the June 1986 issue of SIGCSE BULLETIN.

INTRODUCTION

Normalization is a technique for organizing data. If properly done, it can prevent certain classes of errors in entering data in a data base. For example, if each customer is to have exactly one address, normalization would prevent a data entry clerk from entering the same customer with two different addresses.

Unfortunately, the subject is so new that it has been presented in different inequivalent versions in many of the leading textbooks. This has resulted in a great deal of confusion.

We hope to alleviate this confusion by explaining exactly what the differences in presentation are. We also present an algorithmic method which is easy to use to introduce students to the subject.

In most elementary data base management textbooks, normalization for "second", "third" and "Boyce-Codd" normal forms is treated in a non-algorithmic way, requiring a human being to recognize a pattern. Then, when this pattern is recognized, a file is split into two smaller files. Then one must reevaluate which facts are true on the smaller files and see if the pattern occurs in either of the smaller files. Few of the elementary textbooks point out the difficulties of finding out which are the true facts on the smaller files.

If the pattern occurs in the smaller files, they must be split into still smaller files. This continues until the pattern no longer appears. It is easy to make a mistake using this method.

Furthermore, the definition of "third normal form" differs dramatically from one textbook to another. Many textbooks add bizarre and unnatural hypotheses. Yet one definition, in the graduate text by Ullman [8], is accepted by the data base research community. This accepted definition is also the easiest to state.

In addition, a well-known algorithm, invented by Bernstein [1], is available for decomposing files into the third normal form of the accepted definition. It is actually not necessary to know the definition of third normal form in order to perform the algorithm.

This article begins with an example to illustrate the uses of normalization, then gives an outline of the Bernstein algorithm. After the outline, the algorithm is explained in detail. Then the accepted definition of third normal form is stated, and it is demonstrated that many popular elementary textbooks have different inequivalent definitions. The hidden difficulties of the recognize and split method are explained in the last section.

WHY NORMALIZE

Suppose an alumni office at a large university has a file of records of donations. The record type has the following fields:

NAME STREET CITY STATE CLASS-GRAD AMOUNT-DONATED DATE-DONATED

Among the record instances are:

Kissinger,H.	100 Mass Ave	Cambridge	Ma	1951	$ 4000	Jan 1979
Kissinger,H.	100 Mass Ave	Cambridge	Ma	1951	$ 5000	June 1980
Kissinger,H.	100 Mass Ave	Cambridge	Ma	1951	$ 5000	Jan 1983
O'Connor, S.	14 Forest Dr.	Bethesda	Md	1955	$ 10000	Jan 1981

This is a poor design. First, note the REDUNDANCY. The first 3 record instances repeat the same name, address, and graduation class. But there are worse problems with this design.

Suppose Mr. H. Kissinger calls the alumni office and says that he has moved to Washington, D.C., and leaves his new address. Then suppose only the first record instance (Jan 1979) is updated. At this point, the data base is INCONSISTENT.

Now suppose someone decides to trim the data base by discarding all records of donations before Jan 1982. Then both Kissinger's new (correct) address and O'Connnor's address are lost. When it comes time to ask for more money, two excellent past donors will be overlooked.

Perhaps the addresses should be kept in a different file from the donation records. For example, there might be two files, one with record field names:

NAME STREET CITY STATE CLASS-GRAD

and record instances:

Kissinger, H.	57 Main	Washington	D.C.	1951
O'Connor,S	14 Forest Dr	Bethesda	Md.	1955

and another file with record field names:

NAME AMOUNT-DONATED DATE-DONATED

and record instances:

Kissinger, H.	$ 4000	Jan 1979
Kissinger,H.	$ 5000	Jun 1980
Kissinger, H.	$ 5000	Jan 1983
O'Connor, S.	$ 10000	Jan 1980

Then any updating and changing of the donations file would not affect the address file, and one would have more confidence in the design of the data base. The process which assures this confidence is called normalization.

OUTLINE OF THE ALGORITHM

The easiest method for normalization is Bernstein's algorithm. An outline is given first, and then the algorithm is given in detail.

Step 1: Find out the facts about the real world.

This is a difficult step and must be done in the design of a data base no matter what normalization method is used. This takes more time than all the other steps put together. The result is a list of future record field names and the many-to-one or "functional" relationships between the fields with those names.

In the "recognize and split" method the "facts" must ALSO be established for each new smaller file. Deriving the facts for the smaller files from those of the larger file is not easy. The facts, or functional dependencies, need only be determined ONCE for the Bernstein method.

Step 2: reduce the list of functional relationships

This is a straightforward polynomial time algorithm, explained below. It can easily be done with pencil and paper on a small list, and can be programmed for a big list.

Step 3: find the "keys"

This is a difficult step which must also be done for the old "recognize and split" method. It need only be done once in the Bernstein method. In "recognize and split" it must be done for each new smaller file in order to recognize the pattern in the new smaller file.

Step 4: Combine functional relationships with the same left hand side, make a new smaller file for each one, add a key file if no file contains any key, and eliminate files which are contained in other files.

This is an easy step.

The detailed explanation of Bernstein's algorithm begins with a discussion of functional dependence.

SOC-SEC-NUM	LAST-NAME	FIRST-NAME	DATE-OF-BIRTH
111-11-7777	Brown	Mary	1-15-1950
222-22-7777	Brown	Alice	6-7-1940
999-99-7777	Davis	Alice	6-7-1940
666-66-7777	Davis	George	2-2-1945
333-33-7777	Jones	George	12-12-1960
777-77-9999	Jones	Mary	12-12-1960

SOC-SEC-NUM -- > LAST-NAME, FIRST-NAME, DATE-OF-BIRTH
LAST-NAME, FIRST-NAME -- > SOC-SEC-NUM

Figure 1: Functional Dependency

FUNCTIONAL DEPENDENCE

The first step in normalization is to find out the facts about the real world. These facts must be organized into lists of FUNCTIONAL DEPENDENCIES. This must be done for any normalization method. First, make a list of what will be the future record field names. These future record field names are called ATTRIBUTES in data base literature. Then, categorize the relationships between the record fields, or attributes in terms of functional dependencies.

For example, every person lives on one street. One says that the attribute STREET is FUNCTIONALLY DEPENDENT on the attribute NAME. This means, in a record instance, once a NAME is specified, there is only one possible value for the STREET. The following are equivalent ways of saying this:

1. STREET FUNCTIONALLY DEPENDS on NAME

2. NAME FUNCTIONALLY DETERMINES STREET

3. NAME $\Rightarrow$ STREET

4. There is a many - one relationship between NAME and STREET (Many people may live on one STREET).

In general, one speaks of collections or sets of attributes functionally determining other collections of attributes. But this really describes the relationships possible among the values for those attributes in the record instances in the data base. Another example of functional dependencies is given in figure 1.

The first step of the Bernstein algorithm is to determine the attributes of the data to be stored in the data base, and the functional dependencies between collections of attributes. Then one must make that list minimal, by getting rid of dependencies which

can be deduced from other dependencies. Fortunately, there is a relatively easy algorithm for doing this.

RIGHT HAND SIDE ONE ATTRIBUTE

To minimize the list of functional dependencies, the first step is to rewrite that list of dependencies so that the right hand side of each dependency is exactly one attribute. This is easy, since the definition of functional dependency implies that if X ⇒ Y,Z then X ⇒Y and X⇒Z.

Thus

NAME ⇒ STREET,CITY, STATE, CLASS-GRAD

can be rewritten

NAME ⇒ STREET

NAME ⇒ CITY

NAME ⇒ STATE

NAME ⇒ CLASS-GRAD.

X-CLOSURE

In order to do the next step in minimizing the list of functional dependencies, one needs the X-CLOSURE ALGORITHM.

X-CLOSURE ALGORITHM

TO FIND WHICH ATTRIBUTES ARE FUNCTIONALLY DEPENDENT ON X, A GIVEN SET OF ATTRIBUTES.

STEP 1: Let X(0) = X. Let N = 0.

STEP 2: If there is a dependency A ⇒ B whose left side (A) is contained in X(N), but whose right side (B) is not in X(N), then add B to X(N) to make X(N + 1). That is, X(N + 1) is the union of X(N) and B.

Increment N and repeat this step until no new attributes can be added.

Let us illustrate this algorithm with the example at the beginning of the paper:

NAME ⇒ STREET,CITY,STATE,CLASS-GRAD

NAME,DATE-DONATED ⇒ AMOUNT-DONATED

Suppose "X" is NAME, DATE-DONATED (i.e., the goal is to find all attributes functionally dependent on NAME,DATE-DONATED) First, let

X(0) = NAME,DATE-DONATED.

Then, looking at the first dependency, the left hand side is contained in X(0), so we may add the right hand side to obtain X(1). Thus

X(1) = NAME,DATE-DONATED,STREET, CITY,STATE,CLASS-GRAD.

The left hand side of the second dependency is contained in X(1), so its right hand side may be adjoined. Thus obtain:

X(2) = NAME, DATE-DONATED, STREET, CITY, STATE, CLASS-GRAD, AMOUNT-DONATED.

This time there is no dependency whose right hand side is not in X(2), so terminate.

REDUNDANT DEPENDENCIES

The X-closure algorithm is needed in both of the next two steps for minimalizing the list of dependencies. First one must get rid of redundant dependencies as follows:

1. Take each dependency in turn. Look at a dependency X ⇒ Y, where X and Y are sets of attributes.

2. Make a REDUCED set of dependencies by taking X ⇒ Y out of the list of all dependencies. Find the closure of X in the REDUCED list. If that closure contains Y, then X ⇒ Y is redundant.

Here is an example:

PHARMACY-ACCOUNT-NUMBER ⇒ PATIENT-ID

PATIENT-ID ⇒ DOCTOR-ID

PHARMACY-ACCOUNT-NUMBER,DRUG -⇒ DOCTOR-ID

Let X = X(0) = PHARMACY-ACCOUNT-NUMBER,DRUG. Then look only at the reduced set of dependencies:

PHARMACY-ACCOUNT-NUMBER ⇒ PATIENT-ID

PATIENT-ID ⇒ DOCTOR-ID.

Then X(1) = PHARMACY-ACCOUNT-NUMBER,DRUG,PATIENT-ID since the left hand side of the first dependency is contained in X(0), enabling us to add on the right hand side.

Next, using the second dependency, get X(2) = PHARMACY-ACCOUNT-NUMBER, DRUG, PATIENT-ID and DOCTOR-ID.

Thus the closure of DRUG, PHARMACY-ACCOUNT-NUMBER in

PHARMACY-ACCOUNT-NUMBER ⇒ PATIENT-ID

PATIENT-ID ⇒ DOCTOR-ID

includes DOCTOR-ID, so the dependency

PHARMACY-ACCOUNT-NUMBER,DRUG ⇒DOCTOR-ID is redundant and can be eliminated.

MINIMAL LEFT HAND SIDE

The next step in reducing the list of dependencies is to make sure the left hand side is minimal for each dependency. This is a tedious but straightforward algorithm:

Eliminate one of the attributes, A on the left hand side of one of the dependencies. Look at the remainder, R, of the attributes on the left hand side of that dependency. Find the closure of R in the ORIGINAL set of dependencies. If that closure contains the right hand side of the dependency in question, then the attribute A may be eliminated.

Perform this check on each attribute of the left hand side of each dependency.

Here is an example:

LAST-NAME, SOCSECNO ⇒ FIRST-NAME

SOCSECNO ⇒ LAST-NAME

In this example, the closure of SOCSECNO contains FIRST-NAME and LAST-NAME Thus LAST-NAME is not needed in the first dependency, and this list can be replaced with

SOCSECNO ⇒ FIRST-NAME

SOCSECNO ⇒ LAST-NAME.

When both the redundant dependencies and the non-minimal left hand sides have been eliminated, the list is minimal and it is easier to find the keys.

KEYS

A KEY for a file of records (or a relation) is a collection or set X of attributes which

(1) functionally determines all attributes in the related record type and

(2) has no proper subset with this property.

Clearly, the set of all attributes of a record type has the first property. It is the MINIMALITY requirement of the second property which makes this a useful concept. However, some texts do not include minimality in their definition of key. These texts refer to "minimal keys" to denote what is called simply a key in this paper.

Assume none of the files have duplicate records. This implies that duplicate values for a key (in record instances) are not allowed. This means the usual file concept of "secondary key" (which allows look-up on attributes which may have duplicate values) does not satisfy the data base definition of "key".

However, this does not mean that there cannot be two different keys for a given file of records. Some of the database textbooks make a distinction between "primary keys" and "candidate keys", where the candidate key is just the definition given above for key and a primary key is one of the candidate keys that the user has chosen (perhaps for sorting the record instances in the file, or for making a no-duplicates allowed index.)

Suppose we have the following dependencies:

SOCSECNO $\Rightarrow$ LAST-NAME,FIRST-NAME,DATE-OF-BIRTH

LAST-NAME,FIRST-NAME $\Rightarrow$ SOCSECNO

Then the record type with attributes

SOCSECNO,LAST-NAME,FIRST-NAME,DATE-OF-BIRTH

has two keys. SOCSECNO is a key , and LAST-NAME,FIRST-NAME is a key. No two record instances have the same LAST-NAME and the same FIRST-NAME. No two record instances have the same SOCSECNO.

To determine whether or not a given collection of attributes X is a key,

(1) Use the X-CLOSURE ALGORITHM of the previous section to see if all the attributes of the record type are functionally dependent on the given set X.

(2) If the set X has more than one attribute, make sure that no proper subset of X also has this property, by checking proper subsets of X using the same closure algorithm . In fact, one can make this a manageable process by elimination of a single attribute of X at a time. If any of the subsets thus obtained still determine all of the attributes, then X is not minimal.

The work involved in searching for all the keys can be shortened by the following two observations:

(1) If an attribute is never on the left hand side of a dependency, it is not in any key, unless it is also never on the right.

(2) If an attribute is never on the right hand side of a dependency, then it must be in every key.

As an example, suppose the attributes

PHARMACY-ACCOUNT-NUMBER,PATIENT-ID,DOCTOR-ID,DRUG,QUANTITY

satisfy the dependencies:

PHARMACY-ACCOUNT-NUMBER $\Rightarrow$ PATIENT-ID

PATIENT-ID $\Rightarrow$ DOCTOR-ID

SOC-SEC-NUM -- > LAST-NAME
SOC-SEC-NUM -- > FIRST-NAME
SOC-SEC-NUM -- > DATE-OF-BIRTH
LAST-NAME, FIRST-NAME -- > SOC-SEC-NUM

1. On right but not on left:

DATE-OF-BIRTH
(In no key)

2. Not on right:

(None)
(In all keys)

3. Check subsets containing attributes which must be in all keys (start with smallest):

(a) SOC-SEC-NUM	is a key
(b) LAST-NAME	not a key
(c) FIRST-NAME	not a key
(d) LAST-NAME, FIRST-NAME	is a key

Figure 2: Keys

PATIENT-ID,DRUG $\Rightarrow$ QUANTITY

The following cannot be in keys:

DOCTOR-ID,QUANTITY.

The following must be in every key:

PHARMACY-ACCOUNT-NUMBER, DRUG.

To show that PHARMACY-ACCOUNT-NUMBER, DRUG is in fact a key, one can use the X-closure algorithm. This will show that PHARMACY-ACCOUNT-NUMBER, DRUG functionally determines all the attributes. No subset can have this property, since all keys must contain these attributes.

In the general case, as illustrated in figure 2, one would have to try adding various other attributes to those which must be in every key until all the keys were obtained. This is a difficult step, but this step is necessary in the old recognize and split method as well.

After finding the keys, the last easy step of Bernstein's algorithm can be performed.

THE LAST STEP OF THE ALGORITHM

The last step begins by combining any dependencies with the same left hand side. This

is allowed since if X functionally determines Y, and X functionally determines Z, then X functionally determines the attributes in Y,Z combined.

Now make one file for each dependency, containing all of the attributes of both sides of that dependency. Then if no key for the original file is in any of these new files, add a file with all the attributes from some key.

If some file contains all the attributes of some other file, then eliminate the smaller file. This completes Bernstein's algorithm.

You are guaranteed what is called a "lossless join", "dependency preserving" decomposition into third normal form.

LOSSLESS JOIN means that any record instances which were in the original file can be reconstructed from the new smaller files. This could not happen, for instance, if we had split the files up in such a way that there was no overlap in the attributes. The lossless join is guaranteed by the existence of a file containing one of the keys. The old recognize and split method also guarantees lossless join.

Dependency preserving means you can use no-duplicates-allowed indexes to enforce the dependencies which are true in the real world. This means someone who tries to enter two addresses or two salaries for the same person can be prevented from doing so by the software. This property is NOT guaranteed by the recognize and split method.

The technical definitions of lossless join and dependency preserving can be found in [8]. It is also proved there that this algorithm guarantees those properties and produces third normal form files.

An illustration of the entire Bernstein algorithm is given in figure 3.

THIRD NORMAL FORM

Now that the easiest way to decompose a file into third normal form smaller files has been explained, it is time to give the standard, accepted definition for third normal form. First define PRIME attributes.

A PRIME attribute is one which is in some key. A NON-PRIME attribute is in no key.

THIRD NORMAL FORM

A file is in THIRD NORMAL FORM if whenever X functionally determines a NON-PRIME attribute A, and A is not contained in X, then X contains a key.

In order to test whether or not a file of records is in third normal form, one must have a list of keys and a list of dependencies. Check out each attribute A which is in no key.

List all functional dependencies. Write with one attribute on the right hand side.	PHARMACY-NUM -- > PATIENT-ID PATIENT-ID -- > DOCTOR-ID PHARMACY-NUM, DRUG -- >DOCTOR-ID PATIENT-ID, DRUG -- > QUANTITY	LAST-NAME, SOC-SEC-NUM -- >FIRST-NAME SOC-SEC-NUM -- > LAST-NAME SOC-SEC-NUM -- > DATE-OF BIRTH LAST-NAME, FIRST-NAME -- >SOC-SEC-NUM
2(a) Remove each dependency one by one to form *reduced* list. Check closure of left hand side in *reduced* list. If it contains right hand side, then removed dependency was redundant.	PHARMACY-NUM, DRUG -- > DOCTOR-ID was redundant	none redundant
2(b) If left hand side has more than one attribute, remove one attribute at a time and check closure of remaining attributes in *original* list. If it contains right hand side, removed attribute was not needed.	left hand sides of remaining dependencies all minimal	LAST-NAME, SOC-SEC-NUM -- > FIRST-NAME can be replaced with SOC-SEC-NUM -- >FIRST-NAME
3. Find keys	PHARMACY-NUM, DRUG	(1) LAST-NAME, FIRST-NAME and (2) SOC-SEC-NUM
4(a) combine dependencies with same left hand side; make files. (b) add key file if needed (c) eliminate files whose attributes are all contained in another file.	four files: 1. PHARMACY-NUM, PATIENT-ID 2. PATIENT-ID, DOCTOR-ID 3. PATIENT-ID, DRUG, QUANTITY 4. PHARMACY-NUM, DRUG	one file: SOC-SEC-NUM, LAST-NAME, FIRST-NAME, DATE-OF-BIRTH

Figure 3: The Bernstein Algorithm

(i.e. A is non-prime.) Is A on the right but not the left hand side of some dependency? If so, does the left hand side contain a key? If the left hand side does not contain a key, then the file is not in third normal form.

One way a set of attributes X could NOT contain a key would be if X were properly contained in a key. (This is the basis for the definition of "second normal form" , largely of historical interest: A file is not in SECOND NORMAL FORM if X $\Rightarrow$ A, X does not contain A, A is non-prime, and X is a proper subset of a key.)

Another way X could NOT contain a key, would be if there were some non-prime attributes in X and X neither contained a key, nor was contained in a key. This happened with the dependency

PATIENT-ID,DRUG $\Rightarrow$ QUANTITY

in the example

PHARMACY-ACCOUNT-NUMBER $\Rightarrow$ PATIENT-ID

PATIENT-ID $\Rightarrow$ DOCTOR-ID

PATIENT-ID,DRUG $\Rightarrow$ QUANTITY

where the key was PHARMACY-ACCOUNT-NUMBER,DRUG.

There has been a great deal of variation in the definition of third normal form. One of the most widely used textbooks for computer science undergraduates, C.J. Date's "An Introduction to Database Systems"[3] restricts the definition of third normal form to files which have only one key. There is no such restriction in Ullman's definition. Furthermore, it is difficult to show that the definitions are equivalent in the restricted case, as they are stated very differently.

Date also makes the assumption that all functional dependencies have the left hand side minimal. This he calls a "full functional dependency". This makes a difference in his definitions as we shall see. Ullman does not make this assumption.

Date first defines third normal form as follows (this is from the fourth edition, page 366-377) :

"A relation R is in 3NF if and only if the nonkey attributes of R(if any) are

a) mutually independent

and

b) fully dependent on the primary key of R."

A "relation" is just another word for a file of equal length records.

Date describes full dependence on a key as meaning not dependent on a proper subset of a key. A "non-key" attribute is the same as a "non-prime" attribute. That is, it is not in any key. Since Date is assuming only one key, that is what is meant by "the primary key".

By "mutually independent" Date means "none of the attributes concerned is functionally dependent on any of the others". Remember that this is full functional dependency. In the third edition, it is pointed out that "attribute" can be "composite", or what I have called a set of attributes. It is not clear if a set of attributes which contains some non-prime and some prime components would be a "non-key (composite) attribute".

Then, a second definition is given:

"A relation R is in third normal form (3NF) if and only if for all time, each tuple of R consists of a primary key value that identifies some entity, together with a set of mutually independent attribute values that describe that entity in some way."

A "tuple" is a record instance. An "entity", in this case, is some object in the real world.

Finally, Date gives a third definition of third normal form:

"A relation is in third normal form (3NF) if and only if it is in 2NF [second normal form] and every nonkey attribute is nontransitively dependent on the primary key"

Let us give some examples to show that "transitive dependency" must be very carefully defined in order to make this definition correct.

First, from the definition of functional dependency, If A⇒B and B⇒C, then A⇒C. This is the transitivity property of functional dependencies.

Now look at the file

NAME, STREET,CITY,STATE, YEAR-GRAD

with dependency

NAME ⇒ STREET,CITY,STATE,YEAR-GRAD.

This file is in third normal form. It has one key. Yet one can make up a lot of transitive dependencies which are true and do not violate the standard definition of third normal form. For example,

1. NAME,STREET ⇒ NAME and NAME⇒CITY.

2. NAME ⇒ NAME,CITY and NAME,CITY-⇒STREET.

One only has to make sure that the single key is contained in the left hand side of each dependency. This shows that Date's assumption of minimal left hand side must be included in the definition of transitivity in order for his definition to make sense.

Now look carefully at this transitive dependency:

3. NAME ⇒ NAME and NAME ⇒ STREET.

This has minimal left hand side, only one "primary" key and yet it clearly does not violate Ullman's definition of third normal form. This shows that the two definitions are not equivalent, even when there is only one key.

In addition, as illustrated in figure 4, the assumption of one key only is also essential for this definition to make sense. Files with more than one key always have non-trivial transitive dependencies.

Ullman does not require one key only for his definition of third normal form. The Bernstein third normal form decomposition does not require one key only.

Cardenas[2] quotes Date, Codd and Sharman, and ends up calling Boyce-Codd and third normal form the same, rejecting Date's definition. That is, Cardenas says that if X⇒ Y and Y is not contained in X, then X must contain a key. This is the usual definition of Boyce-Codd normal form. (The NON-PRIME condition is left out). It has been proven that it is not always possible to make a lossless join dependency preserving decomposition into Boyce Codd normal form. This is why Boyce-Codd was thought to be too strong, and third normal form was invented.

We also remark that some authors, for example, Maier[4] and Tsichritzis and Lochovsky[7] have made a more restricted definition of "transitive dependency" , where if K determines X and X determines A, A non-prime, and A not contained in X, then X is

File with dependencies:

SOC-SEC-NUM -- > LAST-NAME
SOC-SEC-NUM -- > FIRST-NAME
SOC-SEC-NUM -- > DATE-OF-BIRTH
LAST-NAME, FIRST-NAME -- > SOC-SEC-NUM

1. Satisfies the accepted definition of third normal form:

If X -- > A, A non-prime, A not contained in X, then X contains a key.

2. Has transitive dependencies:

LAST-NAME, FIRST-NAME -- > SOC-SEC-NUM
SOC-SEC-NUM -- > DATE-OF-BIRTH

Figure 4: Third normal form is NOT the same as "no transitive dependencies".

not allowed to functionally determine K, (hence "X" contains no key). With this definition of RESTRICTED TRANSITIVE DEPENDENCY it is true that NO RESTRICTED TRANSITIVE DEPENDENCY is the same as third normal form. That is, although the definitions in Maier[4] and Tsichritzis and Lochovsky[7] look different, they are equivalent to that in Ullman.

Martin[5], however, says " Suppose A,B and C are three attributes or distinct collections of attributes of a relation R. If C is functionally dependent on B and B is functionally dependent on A, then C is functionally dependent on A. If the inverse mapping is nonsimple(i.e. if A is not functionally dependent on B or B is not functionally dependent on C) then C is said to be TRANSITIVELY dependent on A." Martin then says " A relation is in third normal form if it is in second normal form and every nonprime attribute of R is nontransitively dependent on each candidate key of R."

If "C" in Martin's definitions is the non-prime attribute, and both "A" and "B" contain keys, then B is not functionally dependent on C, and C is "transitively" dependent on A for Martin. However, this condition does not violate the standard definition for third normal form. This also shows that Martin's definition is inconsisent with Ullman's.

THE RECOGNIZE AND SPLIT METHOD

The books by Martin [5], Date [3], Cardenas[2], and Tsichritzis and Lochovsky[7] do not give the Bernstein algorithm. Instead, the recognize and split method is used. The main problem of this method is that one must find out which functional dependencies are true on the new smaller files, and find keys for them before recognizing violations of third normal form on them. This problem is not even mentionned in these books. Let us give

an example.

Suppose one starts with the file of records with attributes PHARMACY-ACCOUNT-NUMBER, PATIENT-ID, DOCTOR-ID,DRUG, QUANTITY and dependencies

PHARMACY-ACCOUNT-NUMBER ⇒ PATIENT-ID

PATIENT-ID ⇒ DOCTOR-ID

PATIENT-ID,DRUG ⇒ QUANTITY

where the key was PHARMACY-ACCOUNT-NUMBER,DRUG.

In this example, the first dependency violates the definition of third normal form. Split the file into two smaller files:

PHARMACY-ACCOUNT-NUMBER, PATIENT-ID,

containing all the attributes of a "bad" dependency, and

PHARMACY-ACCOUNT-NUMBER, DOCTOR-ID,QUANTITY, DRUG

containing every attribute except that on the right hand side of the "bad" dependency.

It is not obvious that the derived dependencies:

PHARMACY-ACCOUNT-NUMBER ⇒ DOCTOR-ID

PHARMACY-ACCOUNT-NUMBER,DRUG ⇒ QUANTITY

are true on the new second file. The fact that the derived dependencies must be considered is not even mentionned in these books. This makes the recognize and split method very much more difficult than the Bernstein algorithm.

In order to do the recognize and split method correctly, one would have to find the closures of each subset of attributes of each new file in order to see what dependencies were true on the new files. Then the keys for each new file must be calculated from the new dependencies. Then "bad" dependencies must be recognized.

Also the recognize and split method does not guarantee preservation of dependencies. Once the original file has been split into

PHARMACY-ACCOUNT-NUMBER, PATIENT-ID,

and

PHARMACY-ACCOUNT-NUMBER, DOCTOR-ID,QUANTITY, DRUG

there is no way one can use the no-duplicates allowed indexing of a file system to enforce the dependency

PATIENT-ID ⇒ DOCTOR-ID.

I.e., one could not prevent by the use of indexes the entering of two doctor id's for the same patient. With the Bernstein algorithm, this would not be a problem.

SUMMARY

It has been shown that the Bernstein method for third normal form is easier than the recognize and split method. Bernstein guarantees third normal form (which implies second normal form), lossless join and preservation of dependecies.

In addition, the confusion about the definition of third normal form in most elementary textbooks has been pointed out. Also, it was shown that the main difficulty of the recognize and split method, that of finding out what functional dependencies are true on the smaller files, is not mentionned in these books.

It is possible and neccessary to use rigor in normalization. What has not been acknowledged, is that the rigor required is actually less difficult than attempting to solve the problem non-algorithmically.

My colleagues and I have had no difficulty in teaching the Bernstein method to undergraduate computer science students at Northeastern University. The students uniformly prefer the Bernstein algorithm to the recognize and split method, which we teach only for Boyce-Codd normal form. They anxiously ask us if there will be any questions on the recognize and split method on the exams.

ACKNOWLEDGEMENT

I would like to thank Professor Patrick Fischer of Vanderbilt University for his comments on the treatment of normalization in the manuscript of my book[6], which led to some of the points in this article.

REFERENCES

1.Bernstein, P. A., "Synthesizing third normal form relations from functional dependencies", ACM Transactions on Database Systems, 1,4(1976), 277-298.

2. Cardenas, A. F., Data Base Management Systems, Allyn and Bacon, Inc.,

Boston, 1985.

3. Date C. J., An Introduction to Database Systems, Addison Wesley, Reading, Massachusetts, 1986.

4. Maier, D., The Theory of Relational Databases, Computer Science Press, Rockville, Maryland, 1983.

5. Martin, J., Computer Data-Base Organization, Prentice-Hall, New Jersey, 1977.

6. Salzberg, B., An Introduction to Database Design, Academic Press, Orlando, Florida, 1986.

7. Tsichritzis, D. C., and Lochovsky, F. H., Data Base Management Systems, Academic Press, New York, 1977.

8. Ullman, J.D., Principles of Database Systems, Computer Science Press, Rockville, Maryland, 1982.