Normalization-Part II

Hi! We are going to continue with Normalization. I hope the basics of normalization are clear to you.

Boyce-Codd Normal Form (BCNF)

The relation *student(sno, sname, cno, cname)* has all attributes participating in candidate keys since all the attributes are assumed to be unique. We therefore had the following candidate keys:

(sno, cno)
(sno, cname)
(sname, cno)
(sname, cname)

Since the relation has no non-key attributes, the relation is in 2NF and also in 3NF, in spite of the relation suffering the problems that we discussed at the beginning of this chapter.

The difficulty in this relation is being caused by dependence within the candidate keys. The second and third normal forms assume that all attributes not part of the candidate keys depend on the candidate keys but does not deal with dependencies *within* the keys. BCNF deals with such dependencies.

A relation R is said to be in BCNF if whenever $X \rightarrow A$ holds in R, and A is not in X, then X is a candidate key for R.

It should be noted that most relations that are in 3NF are also in BCNF. Infrequently, a 3NF relation is not in BCNF and this happens only if

(a) the candidate keys in the relation are composite keys (that is, they are not single attributes),

(b) there is more than one candidate key in the relation, and
(c) the keys are not disjoint, that is, some attributes in the keys are common.

The BCNF differs from the 3NF only when there are more than one candidate keys and the keys are composite and overlapping. Consider for example, the relationship

enrol (sno, sname, cno, cname, date-enrolled)

Let us assume that the relation has the following candidate keys:

(sno, cno)
(sno, cname)
(sname, cno)
(sname, cname)

(we have assumed *sname* and *cname* are unique identifiers). The relation is in 3NF but not in BCNF because there are dependencies

sno → sname
cno → cname

where attributes that are part of a candidate key are dependent on part of another candidate key. Such dependencies indicate that although the relation is about some entity or association that is identified by the candidate keys e.g. *(sno, cno)*, there are attributes that are not about the whole thing that the keys identify. For example, the above relation is about an association (enrolment) between students and subjects and therefore the relation needs to include only one identifier to identify students and one identifier to identify subjects. Providing two identifiers about students *(sno, sname)* and two keys about subjects *(cno, cname)* means that some information about students and subjects that is not needed is being provided. This provision of information will result in repetition of information and the anomalies that we discussed at the beginning of this chapter. If we wish to include further information about students and courses in the database, it should

not be done by putting the information in the present relation but by creating new relations that represent information about entities *student* and *subject*.

These difficulties may be overcome by decomposing the above relation in the following three relations:

(sno, sname)
(cno, cname)
(sno, cno, date-of-enrolment)

We now have a relation that only has information about students, another only about subjects and the third only about enrolments. All the anomalies and repetition of information have been removed.

So, a relation is said to be in the BCNF if and only if it is in the 3NF and every non-trivial, left-irreducible functional dependency has a candidate key as its determinant. In more informal terms, a relation is in BCNF if it is in 3NF and the only determinants are the candidate keys.

DESIRABLE PROPERTIES OF DECOMPOSITIONS

So far our approach has consisted of looking at individual relations and checking if they belong to 2NF, 3NF or BCNF. If a relation was not in the normal form that was being checked for and we wished the relation to be normalized to that normal form so that some of the anomalies can be eliminated, it was necessary to decompose the relation in two or more relations. The process of decomposition of a relation *R* into a set of relations $R_1, R_2, \ldots R_n$ was based on identifying different components and using that as a basis of decomposition. The decomposed relations $R_1, R_2, \ldots R_n$ are projections of *R* and are of course not disjoint otherwise the glue holding the information together would be lost. Decomposing relations in this way based on a recognize and split method is not a particularly sound approach since we do not even have a basis to determine that the original relation can be constructed if necessary from the decomposed relations. We now

discuss desirable properties of good decomposition and identify difficulties that may arise if the decomposition is done without adequate care. The next section will discuss how such decomposition may be derived given the FDs.

Desirable properties of decomposition are:

1. Attribute preservation
2. Lossless-join decomposition
3. Dependency preservation
4. Lack of redundancy

We discuss these properties in detail.

Attribute Preservation

This is a simple and an obvious requirement that involves preserving all the attributes that were there in the relation that is being decomposed.

Lossless-Join Decomposition

In these notes so far we have normalized a number of relations by decomposing them. We decomposed a relation intuitively. We need a better basis for deciding decompositions since intuition may not always be correct. We illustrate how a careless decomposition may lead to problems including loss of information.

Consider the following relation

enrol (sno, cno, date-enrolled, room-No., instructor)

Suppose we decompose the above relation into two relations *enrol1* and *enrol2* as follows

enrol1 (sno, cno, date-enrolled)
enrol2 (date-enrolled, room-No., instructor)

There are problems with this decomposition but we wish to focus on one aspect at the moment. Let an instance of the relation *enrol* be

sno	cno	date-enrolled	room-No.	instructor
830057	CP302	1FEB1984	MP006	Gupta
830057	CP303	1FEB1984	MP006	Jones
820159	CP302	10JAN1984	MP006	Gupta
825678	CP304	1FEB1984	CE122	Wilson
826789	CP305	15JAN1984	EA123	Smith

and let the decomposed relations *enrol1* and *enrol2* be:

sno	cno	date-enrolled
830057	CP302	1FEB1984
830057	CP303	1FEB1984
820159	CP302	10JAN1984
825678	CP304	1FEB1984
826789	CP305	15JAN1984

date-enrolled	room-No.	instructor
1FEB1984	MP006	Gupta
1FEB1984	MP006	Jones
10JAN1984	MP006	Gupta
1FEB1984	CE122	Wilson
15JAN1984	EA123	Smith

All the information that was in the relation *enrol* appears to be still available in *enrol1* and *enrol2* but this is not so. Suppose, we wanted to retrieve the student numbers of all students taking a course from *Wilson*, we would need to join *enrol1* and *enrol2*. The join would have 11 tuples as follows:

sno	cno	date-enrolled	room-No.	instructor
830057	CP302	1FEB1984	MP006	Gupta
830057	CP302	1FEB1984	MP006	Jones
830057	CP303	1FEB1984	MP006	Gupta
830057	CP303	1FEB1984	MP006	Jones
830057	CP302	1FEB1984	CE122	Wilson
830057	CP303	1FEB1984	CE122	Wilson

(add further tuples ...)

The join contains a number of spurious tuples that were not in the original relation Enrol. Because of these additional tuples, we have lost the information about which students take courses from WILSON. (Yes, we have more tuples but less information because we are unable to say with certainty who is taking courses from WILSON). Such decompositions are called *lossy* decompositions. A *nonloss* or *lossless* decomposition is that which guarantees that the join will result in exactly the same relation as was decomposed. One might think that there might be other ways of recovering the original relation from the decomposed relations but, sadly, no other operators can recover the original relation if the join does not (why?).

We need to analyze why some decompositions are lossy. The common attribute in above decompositions was Date-enrolled. The common attribute is the glue that gives us the ability to find the relationships between different relations by joining the relations together. If the common attribute is not unique, the relationship information is not preserved. If each tuple had a unique value of Date-enrolled, the problem of losing information would not have existed. The problem arises because several enrolments may take place on the same date.

A decomposition of a relation R into relations $R_1, R_2, \ldots R_n$ is called a *lossless-join* decomposition (with respect to FDs F) if the relation R is always the natural join of the

relations $R_1, R_2, \ldots R_n$. It should be noted that natural join is the only way to recover the relation from the decomposed relations. There is no other set of operators that can recover the relation if the join cannot. Furthermore, it should be noted when the decomposed relations $R_1, R_2, \ldots R_n$ are obtained by projecting on the relation *R*, for example R_1 by projection $\pi_1(R)$, the relation R_1 may not always be precisely equal to the projection since the relation R_1 might have additional tuples called the *dangling* tuples.

It is not difficult to test whether a given decomposition is lossless-join given a set of functional dependencies *F*. We consider the simple case of a relation *R* being decomposed into R_1 and R_2. If the decomposition is lossless-join, then one of the following two conditions must hold

$$R_1 \cap R_2 \rightarrow R_1 - R_2$$

$$R_1 \cap R_2 \rightarrow R_2 - R_1$$

That is, the common attributes in R_1 and R_2 must include a candidate key of either R_1 or R_2. How do you know, you have a loss-less join decomposition?

Dependency Preservation

It is clear that decomposition must be lossless so that we do not lose any information from the relation that is decomposed. Dependency preservation is another important requirement since a dependency is a constraint on the database and if $X \rightarrow Y$ holds than we know that the two (sets) attributes are closely related and it would be useful if both attributes appeared in the same relation so that the dependency can be checked easily.

Let us consider a relation *R(A, B, C, D)* that has the dependencies *F* that include the following:

$A \rightarrow B$

$A \rightarrow C$

etc

If we decompose the above relation into *R1(A, B)* and *R2(B, C, D)* the dependency $A \rightarrow C$cannot be checked (or preserved) by looking at only one relation. It is desirable that decompositions be such that each dependency in *F* may be checked by looking at only one relation and that no joins need be computed for checking dependencies. In some cases, it may not be possible to preserve each and every dependency in *F* but as long as the dependencies that are preserved are equivalent to *F*, it should be sufficient.

Let *F* be the dependencies on a relation *R* which is decomposed in relations $R_1, R_2, \ldots R_n$.

We can partition the dependencies given by *F* such that $F_1, F_2, \ldots F_n$. $F_n$are dependencies that only involve attributes from relations $R_1, R_2, \ldots R_n$respectively. If the union of dependencies $F_i$imply all the dependencies in *F*, then we say that the decomposition has preserved dependencies, otherwise not.

If the decomposition does not preserve the dependencies *F*, then the decomposed relations may contain relations that do not satisfy *F* or the updates to the decomposed relations may require a join to check that the constraints implied by the dependencies still hold.

(Need an example) (Need to discuss testing for dependency preservation with an example... Ullman page 400)

Consider the following relation

sub(sno, instructor, office)

We may wish to decompose the above relation to remove the transitive dependency of *office* on *sno*. A possible decomposition is

S1(sno, instructor)
S2(sno, office)

The relations are now in 3NF but the dependency $instructor \rightarrow office$ cannot be verified by looking at one relation; a join of *S1* and *S2* is needed. In the above decomposition, it is quite possible to have more than one office number for one instructor although the functional dependency $instructor \rightarrow office$ does not allow it.

Lack of Redundancy

We have discussed the problems of repetition of information in a database. Such repetition should be avoided as much as possible.

Lossless-join, dependency preservation and lack of redundancy not always possible with BCNF. Lossless-join, dependency preservation and lack of redundancy is always possible with 3NF.

Deriving BCNF

Should we also include deriving 3NF?

Given a set of dependencies *F*, we may decompose a given relation into a set of relations that are in BCNF using the following algorithm. So far we have considered the "recognize and split" method of normalization. We now discuss Bernstein's algorithm. The algorithm consists of

(1) Find out the facts about the real world.
(2) Reduce the list of functional relationships.
(3) Find the keys.
(4) Combine related facts.

Once we have obtained relations by using the above approach we need to check that they are indeed in BCNF. If there is any relation *R* that has a dependency $A \rightarrow B$ and *A* is not a key, the relation violates the conditions of BCNF and may be decomposed in *AB* and *R* - *A*. The relation *AB* is now in BCNF and we can now check if *R* - *A* is also in BCNF. If not, we can apply the above procedure again until all the relations are in fact in BCNF.

- Data Base Management Systems by Alexis Leon, Mathews Leon
- http://databases.about.com/library

Review Questions

1. Explain BCNF
2. Explain attribute preservation and dependence preservation
3. Explain Loss-less join decomposition