Database Design

When designing a database using the relational model, often one has to select among alternative sets of relation schemes. Some choices are more convenient than others for various reasons. The goal of the designer at this point should be to select relation schemes such that database consistency is preserved following any update operations. If the relation schemes are not properly chosen, anomalies can occur after database operations. For example, let us consider the relation scheme SUPPLIER (sname, saddress, sphone, item, price). The problems with this scheme design are as follows:

1. **Redundancy:** The address of the supplier is repeated once for each item supplied.

2. **Update anomalies:** As a consequence of the redundancy, the address for a supplier in one tuple could be updated while leaving it fixed in another. Thus we would not have a unique address for each supplier as we feel intuitively we should.

3. **Insertion anomalies:** We cannot record an address for a supplier if that supplier does not currently supply at least one item. We might put null values in the 'item' and 'price' components of a tuple for that supplier, but then, when we enter an item for that supplier, will we remember to delete the tuple with nulls ?

4. **Deletion anomalies :** The inverse to problem 'Deletion anomalies' is that should we delete all the items supplied by one supplier, we unintentionally lose track of its address.

In this example, all the above problems disappear if we replace SUPPLIER by two relational schemes : SUPPLIER_DTL(sname, saddress, sphone)
ITEM_SUPPLIED(sname, item, price)
The SUPPLIER_DTL relation stores address and phone number of each supplier only once. The second relation keeps items information supplied by suppliers. So one design goal is to represent supplier information by relations that do not create anomalies following tuple add, delete or update operations. For this purpose the relations should be in a suitable normal form.

Functional Dependency

The integrity constraints in relational database system can be broadly classified into two groups : Domain Dependency and Data Dependency.

Domain Dependency restricts admissible domain values of the attributes e.g. "age of an employee is less than 60 years' or 'height of a person is less than 7 feet'.

Data Dependency requires that if some tuples in the database fulfil certain qualities, then either some other tuples must also exist in the database, or some values of the given tuples must be equal.

Between these two types of dependencies, data dependencies have received wider attention as they have greater impact on the design of the database systems. In fact, it is rather easy to verify domain dependency during update or insertion operations, while data dependency will often lead to anomalies unless relation schemes are properly chosen.

In the relational database literature several types of data dependencies such as functional dependency, multivalued dependency, join dependency etc. have been identified. Among these, the concept of functional dependency is most important.

Formally, a functional dependency fd is a statement, $X \rightarrow Y$ (X functionally determines Y) where X and Y are sets of attributes. A relation r satisfies this fd if for all tuples t_1 and t_2 in r, $t_1(X) = t_2(X)$ implies $t_1(Y) = t_2(Y)$. In other words, r cannot have two tuples which agree in their X values, yet disagree in their Y values.

For example let us consider a relation STUDENT having attributes roll_no, name, address,course, grade etc. The following fds hold :

roll_no → address
roll_no course → grade

The underlying semantics of these two fds are – the roll_no of a student is unique and in each course a student gets unique grade.

The only way to determine the functional dependencies that hold for a relation scheme is by careful examination of the meaning of each attribute. In this sense, the dependencies actually capture semantics of the data.

Full functional dependency

The term full functional dependency is used to indicate the minimum set of attributes in the left side of an fd to functionally determine the right hand side. Formally Y is fully functionally dependent on X if

i) Y is functionally dependent on X
ii) Y is not functionally dependent on any proper subset of X.

Let us consider the relation ORDER(ord#, parts, supp_name, supp_add, qty_ordered, price). It can be said that ord# supp_name → supp_add. But it is not full functional dependency. The full fd is supp_name → supp_add.

Logical implication of dependencies

The definition of functional dependency allows new fds to be derived from known fds. Conversely, a given set of fds may contain some redundant fds. An fd is called redundant in the set of fds F, if F - {f} logically implies f. The set of all fds implied by F is called closure of F and is denoted by F^+.

Keys

The concept of key is very important in any file management system. The key is used to uniquely identify a record. There is an analogous concept for relations with functional dependencies. If R (A_1, A_2, . . ., A_n) is a relation scheme with fds F, and X is a subset of { A_1 , A_2, . . ., A_n }, we say X is a key of R if

i) X → A_1 , A_2, . . ., A_n is implied by F
ii) R is not functionally dependent on any proper subset of X i.e. X → A_1 , A_2, . . ., A_n is a full functional dependency.

There may be more than one key for a relation. They are defined as **candidate keys**. Among the candidate keys, the key which is chosen by the designer is termed as **primary key**. The term **super key** is used to denote any superset of a key.

Let us consider a STUDENT relation having attributes reg_no, name, course, dept, sem, roll_no.

Here reg_no of each student is unique. So with the help of reg_no alone a student record can be found out. Alternatively the combination of (course, dept, sem, roll_no) may find out a student record uniquely. Hence there are two candidate keys for identifying a student record. A designer in this context may chose either (reg_no) or (course, dept, sem, roll_no) as primary key. The combination (reg_no, name) may be considered as super key.

Armstrong's axioms for functional dependencies

There are several inference rules which tell us how one or more dependencies imply other dependencies. The set of rules is often called Armstrong's axioms.

Reflexivity If $Y \subseteq X \subseteq U$ then $X \rightarrow Y$ is logically implied by F. This rule gives the trivial dependency.

Augmentation If $X \rightarrow Y$ holds and $Z \subseteq U$ then $XZ \rightarrow YZ$. Here $XZ = X \cup Z$ Here the given dependency $X \rightarrow Y$ might be in F or from F^+.

Transitivity If $X \rightarrow Y$ and $Y \rightarrow Z$ hold, then $X \rightarrow Z$ holds.

There are several other inference rules that follow from Armstrong's axioms.

The union rule $\{ X \rightarrow Y, X \rightarrow Z \} \models X \rightarrow YZ$

The pseudotransitivity rule $\{ X \rightarrow Y, WY \rightarrow Z \} \models XW \rightarrow Z$

The decomposition rule If $X \rightarrow Y$ holds and $Z \subseteq Y$ then $X \rightarrow Z$ holds

Prime and non-prime attributes: Attributes which are parts of any candidate key of relation are called as prime attribute, others are non-prime attributes. For example, name, address in STUDENT relation are prime attributes, others are non-prime attributes.

Computing Closures

Computing F^+ for a set of dependencies F is a time-consuming task. The set of dependencies in F^+ can be large even if F itself is small. Let us consider the set $F = \{ A \rightarrow B_1, A \rightarrow B_2, \ldots, A \rightarrow B_n \}$. Then F^+ includes all the dependencies $A \rightarrow Y$ where Y is a subset of $\{ B_1, B_2, \ldots, B_n \}$. As there are 2^n such sets Y, we could not expect to list F^+ conveniently even for reasonably sized n.

Steps to determine F^+:

- Determine each set of attributes **X** that appears as a left hand side of some FD in F.
- Determine the set $\mathbf{X}^+$ of all attributes that are dependent on **X**, as given.
- In other words, $\mathbf{X}^+$ represents a set of attributes that are functionally determined by X based on F. And, X^+ is called the **Closure of X under F.**

- All such sets of $\mathbf{X}^+$, in combine, form a closure of F.

Attribute Closure: Attribute closure of an attribute set can be defined as set of attributes which can be functionally determined from it.

At the other extreme, computing X^+ for a set of attributes X is not hard; it takes time proportional to the length of all the dependencies in F. There is a simple way (membership algorithm) to compute X^+ . With the help of an example this can be explained.

Let F consist of the following dependencies :

AB → C	D → EG
C → A	BE → C
BC → D	CG → BD
ACD → B	CE → AG

Let X = BD. $X^{(0)}$ = BD. To compute $X^{(1)}$ we look for dependencies that have a left side B, D or BD. There is only one D → EG , so we adjoin E and G to $X^{(0)}$ and make $X^{(1)}$ = BDEG. In this way if we continue till the results of two iterations become equal.

$X^{(2)}$ = BDEG ∪ C = BCDEG { D → EG , BE → C }
$X^{(3)}$ = BCDEG ∪ AG = ABCDEG {C → A, BC → D, CG → BD, CE → AG }

As it contains all the attributes, $X^{(3)} = X^{(4)} = \ldots$ Thus $(BD)^+$ = ABCDEG

Covers of sets of dependencies

Let F and G be sets of dependencies. We say F and G are equivalent if $F^+ = G^+$. When F and G are equivalent, we sometimes say F covers G.

Minimal Cover

F′ is said to be minimal cover of F if:

i) for any f ∈ F′ , F′ - { f } is not a cover of F
ii) for no X → A in F′ and proper subsets Z of X,

((F′ - { X → A }) ∪ { Z → A }) is a cover of F.

Here the first condition implies that no dependency in F′ is redundant. The second condition guarantees that no attributes on any left side is redundant i.e. each fd is a full functional dependency. It should be pointed out that minimal cover of a set of fds is not unique.
Membership algorithm can be used to check each fd whether they can be deduced from the rest of the set.

Normal Forms

There are a number of normal forms. These normal forms guarantee that most of the problems of redundancy and anomalies do not occur.

Let us consider a table as follows :

The personnel data for an automobile shop

Mech no.	Skill no.	Skill cat.	Mech name	Mech age	Shop no	city	Supv.	Prof
92	113	Body	Harry	22	52	Delhi	Jay	3
47	113,55	Body,Engn	John	41	44	Bombay	Chris	5
43	55	Engn	Anand	23	44	Bombay	Chris	6
52	21,28	Axle, Tire	Peter	25	21	Madras	Bob	2

The given set of fds :

mech no → mech name
mech no → mech age
mech no → shop no
mech no → supv
skill no → skill cat
shop no → mech no
shop no → supv
shop no → city
mech no skill no → prof

The problems of the above design are as follows :

If Hary is the only mechanic in Shop Number 52 and he quits, all tuples concerning him will have to be deleted. But at the same time, the information that Jay is the supervisor of Shop no. 52 will be deleted. So we are going to loose one information. There are many other problems as well.

Thus a poor tabular design for a database may cause problems of redundancy and anomalies. Normal forms evolved in the relational model as an attempt to do away with these problems.

First Normal Forms (1NF)

The 1NF has the property that every data entry for each attribute is non-decomposable. In the above table 'skill no.' and 'skill cat' contain decomposable data. So the above table may be converted to 1NF as follows :

Mech no.	Skill no.	Skill cat.	Mech name	Mech age	Shop no	city	Supv.	Prof
92	113	Body	Harry	22	52	Delhi	Jay	3
47	113	Body	John	41	44	Bombay	Chris	5
47	55	Engn	John	41	44	Bombay	Chris	1
43	55	Engn	Anand	23	44	Bombay	Chris	6
52	21	Axle	Peter	25	21	Madras	Bob	2
52	28	Tire	Peter	25	21	Madras	Bob	6

Second Normal Form (2NF)

A relation R is in second normal form if every nonprime attributes of R is fully functionally dependent on each relation key.

One may verify that (mech no skill no) combination of attributes is a valid key for the above relation. Inspecting the set of fds it can be said that many of the non-prime attributes are functionally dependent on only part of the key i.e. either mech no or skill no. So it violates the regulation of 2NF. The table has to be converted into 2NF compatible form as follows :

Mechanic :

mech no	mech name	mech age	shop no	city	supv
92	Harry	22	52	Delhi	Jay
47	John	41	44	Bombay	Chris
43	Anand	23	44	Bombay	Chris
52	Peter	25	21	Madras	Bob

Skill :

skill no	skill cat
113	Body
55	Engn
21	Axle
28	Tire

Proficiency :

mech no	skill no	prof
92	113	3
47	113	5
47	55	1
43	55	6
52	21	2
52	28	6

Another Example

Let us consider the relation FLIGHT (flight#, type-of-aircraft, date, source, destination). The given fds are : flight# → type-of-aircraft
flight# date → source destination

Here (flight#, date) is a key but type-of-aircraft depends only on flight#.

The Third Normal Form (3NF)

A relation is in third normal form if it is in second normal form and no nonprime attribute is functionally dependent on other nonprime attributes.

Alternatively a relation scheme R is in third normal form if whenever X → A holds in R and A is not in X, then either X is a superkey for R, or A is prime.

For the example "personnel data", "Skill" and "Proficiency" relations are already in 3NF as there is no fds related to these relations where nonkey → nonkey fd is there. Whereas in the "Mechanic" relation there are two such fds. They are shop no → supv and shop no → city. So the relation "Mechanic" may be converted to two relations as follows :

Mechanic :

mech no	mech name	mech age	shop no
92	Harry	22	52
47	John	41	44
43	Anand	23	44
52	Peter	25	21

Shop :

shop no	city	supv
52	Delhi	Jay
44	Bombay	Chris
21	Madras	Bob

Another Example of violation of 3NF

Let us consider the relation ORDER (order#, parts, supplier, unit_price, qty) with functional dependencies as follows :
order# → parts supplier qty
supplier parts → unit_price

Here order# is a key, but unit_price depends on non prime attributes supplier and parts. Hence this relation is not in 3NF. In this example there are insertion/deletion anomalies. We can't store here unit_price information for any parts supplied by a supplier unless an order has been placed for that parts.

Boyce-Codd Normal Form (BCNF)

A relation R with fds F is said to be in BCNF if whenever X → A holds in R, and A is not in X, then X is a superkey for R i.e. X is a key or it contains a key.

Example : Let us consider a relation RENT_DUE (qtr#, emp_id, date_of_occupancy, rent) with fds as follows :

qtr# date_of_occupancy → emp_id rent
emp_id → qtr#

Here the first fd implies that at any instant of time a quarters is allocated to only one employee and the rent is fixed at the time of occupancy. The second fd represents the information that an employee is allocated only one quarter. This relation is in 3NF with qtr# and date_of_occupancy as key. This scheme suffers from the anomaly that we cannot record the quarters allocated to an employee unless we know the date of occupancy. To remove such problems the relation should be in BC normal form i.e. BCNF.

Table 1: (emp_id, date_of_occupancy, rent)
Table 2: (emp_id, qtr#)

Another Example:

The following is a college enrolment table with columns `student_id`, `subject` and `professor`.

student_id	subject	professor
101	Java	P.Java
101	C++	P.Cpp
102	Java	P.Java2
103	C#	P.Chash
104	Java	P.Java

In the table above:

- One student can enrol for multiple subjects. For example, student with **student_id** 101, has opted for subjects - Java & C++
- For each subject, a professor is assigned to the student.
- And, there can be multiple professors teaching one subject like we have for Java.

In the table above `student_id, subject` together form the primary key, because using `student_id` and `subject`, we can find all the columns of the table.

One more important point to note here is, one professor teaches only one subject, but one subject may have two different professors.

Hence, there is a dependency between `subject` and `professor` here, where `subject` depends on the professor name.

This table satisfies the **1st Normal form** because all the values are atomic, column names are unique and all the values stored in a particular column are of same domain.

This table also satisfies the **2nd Normal Form** as there is no **Partial Dependency**.

And, there is no **nonkey → nonkey Dependency**, hence the table also satisfies the **3rd Normal Form**.

But this table is not in **Boyce-Codd Normal Form**. Here `student_id, subject` form primary key, which means `subject` column is a **prime attribute**. But, there is one more dependency, `professor` → `subject`.

BCNF conversion

Student

student_id	p_id
101	1
101	2

and so on...

Professor

p_id	professor	subject
1	P.Java	Java
2	P.Cpp	C++

and so on...

Fourth Normal Form (4NF)

The fourth normal form deals with multiple associations among the attributes. Let us take the same example of personnel information of an automobile shop. A shop may have more than one supervisor. If shop number 21 now has two supervisors Bob and Katz, it would be necessary to add 2 additional tuples to the relation. The updated table is shown below:

Mech no.	Skill no.	Skill cat.	Mech name	Mech age	Shop no	city	Supv.	Prof
92	113	Body	Harry	22	52	Delhi	Jay	3
47	113	Body	John	41	44	Bombay	Chris	5
47	55	Engn	John	41	44	Bombay	Chris	1
43	55	Engn	Anand	23	44	Bombay	Chris	6
52	21	Axle	Peter	25	21	Madras	Bob	2
52	28	Tire	Peter	25	21	Madras	Bob	6
52	21	Axle	Peter	25	21	Madras	Katz	2
52	28	Tire	Peter	25	21	Madras	Katz	6

The 4NF deals with such redundancy.

Definition 4NF : A relation scheme R is in 4NF with respect to a set of dependencies F if, for every nontrivial multivalued dependency x ->> y in F^+ , x is a superkey for R.

The following represents the example data base satisfying 4NF.

Mechanic :

mech no	mech name	mech age	shop no
92	Harry	22	52
47	John	41	44
43	Anand	23	44
52	Peter	25	21

Skill :

skill no	skill cat
113	Body
55	Engn
21	Axle
28	Tire

Proficiency :

mech no	skill no	prof
92	113	3
47	113	5
47	55	1
43	55	6
52	21	2
52	28	6

Shop Location

Shop Supervisors

Shop No.	City
52	Delhi
44	Bombay
21	Madras

Shop No.	Supervisor
52	Jay
44	Chris
21	Bob
21	Katz

Rules for 4th Normal Form

For a table to satisfy the Fourth Normal Form, it should satisfy the following two conditions:

1. It should be in the **Boyce-Codd Normal Form**.
2. And, the table shou
3.
4. ld not have any **Multi-valued Dependency**.

What is Multi-valued Dependency?

A table is said to have multi-valued dependency, if the following conditions are true,

1. For a dependency A → B, if for a single value of A, multiple value of B exists, then the table may have multi-valued dependency.
2. Also, a table should have at-least 3 columns for it to have a multi-valued dependency.
3. And, for a relation `R(A,B,C)`, if there is a multi-valued dependency between, A and B, then B and C should be independent of each other.

If all these conditions are true for any relation(table), it is said to have multi-valued dependency.

More examples of 4NF

Example 1

Emp_name	Project_name	Dependent_name
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

Brown	W	Jim
Brown	X	Jim
Brown	Y	Jim
Brown	Z	Jim
Brown	W	Joan
Brown	X	Joan
Brown	Y	Joan
Brown	Z	Joan
Brown	W	Bob
Brown	X	Bob
Brown	Y	Bob
Brown	Z	Bob

In the above table two MVDs are there :

Emp_name -->> Project_name
Emp_name -->> Dependent_name

It is to be noted that Dependent_name is independent of Project_name.
To satisfy 4NF the above table may be split into two as follows:

Emp_Project

Emp_name	Proj_name
Smith	X
Smith	Y
Brown	W
Brown	X
Brown	Y
Brown	Z

Emp_dependents

Emp_name	Dep_name
Smith	Anna
Smith	John
Brown	Jim
Brown	Joan
Brown	Bob

Example 2

Consider a vendor supplying many items to many projects in an organisation.

V_I_P

Vendor_code	**Item_code**	**Project_no.**
V1	I1	P1
V1	I2	P1
V1	I1	P3
V1	I2	P3
V2	I2	P1
V2	I3	P1
V3	I1	P1

V3 I1 P2

Here two MVDs exist -- Vendor_code -->> Item_code
Vendor_code -->> Project_no.
To satisfy 4NF, the above table may be split into two :

Vendor_Supply

Vendor_code	Item_code
V1	I1
V1	I2
V2	I2
V2	I3
V3	I1

(a)

Vendor_Project

Vendor_code	**Project_no**
V1	P1
V1	P3
V2	P1
V3	P1
V3	P2

(b)

These relations (a) & (b) still have a problem. Even though a vendor is capable to supply items, a project may not need the same. For example **V2 is capable to supply I3 but the same is not needed by project P1.**

Multivalued dependencies help us understand and tackle some forms of repetition of information that cannot be understood in terms of functional dependencies. There are types of constraints called **join dependencies** that generalise multivalued dependencies and lead to another normal form called **project-join normal form** (PJNF) or **fifth normal form** (5NF).

So another relation is essential to keep the tables in 5NF.

Project_Items

Project_no	**Item_code**
P1	I1
P1	I2
P2	I1
P3	I1
P3	I2

(c)

Now if vendor_supply and vendor_project relations are joined, the original V_I_P can be constructed. Now the intermediate resultant relation is to be joined with Project_Items the following desired relation is obtained :

modified_V_I_P

Vendor_code	Item_code	Project_no.
V1	I1	P1
V1	I2	P1
V1	I1	P3
V1	I2	P3
V2	I2	P1
V3	I1	P1
V3	I1	P2

A practical problem with the use of the said constraints is that they are not only hard to reason with, but there is also no set of sound and complete inference rules for reasoning about the constraints.

Another Example of 5NF

Let us consider that certain technicians represent certain companies and also specialise in servicing certain models of computer. The following relation **Business** embodies these two constraints.

Business

Technician	Company	Model
Smith J	IBM	PS2
Smith J	IBM	AT
Smith J	IBM	RT
Smith J	Compaq	PC+
Smith J	Compaq	AT
Smith J	Compaq	Portable
Smith J	DEC	RT
Smith J	DEC	Rainbow
Chow D	Compaq	PC+
Chow D	Compaq	AT
Chow D	Compaq	Portable
Chow D	DEC	RT
Chow D	DEC	Rainbow
Chow D	IBM	PS2
Chow D	IBM	AT
Chow D	IBM	RT

The above relation contains two MVDs –

Technician -->> Company
Company -->> Model

It is clear that the relation **Business** contains some redundancy due to the above MVDs and may be reduced to the 4NF by decomposing into two following relations :

Represents

Technician	Company
Smith J	IBM
Smith J	Compaq
Smith J	DEC
Chow D	Compaq
Chow D	DEC
Chow D	IBM

Sells

Company	Model
IBM	PS2
IBM	AT
IBM	RT
Compaq	PC+
Compaq	AT
Compaq	Portable
DEC	RT
DEC	Rainbow

One may verify that **Represents** and **Sells** when joined, reproduces the original relation **Business**

Let us now assume that an additional constraint exists in this enterprise. Not all technicians are trained to service all computer models. The services by technicians are given below :

Chow D -- Portable, Rainbow
Smith J – AT, RT, PS2

Therefore the relation **Business** does not correctly represent these constraints. The solution is to add a MVD – **Technician -->> Model** into the database. This can be achieved if we join **Business** with the relation **Training** (given below).

Training

Technician	Model
Chow D	Portable
Chow D	Rainbow
Smith J	AT
Smith J	RT
Smith J	PS2

Decomposition of Relation Schemes

The decomposition of a relation scheme R = { A_1, A_2, . . ., A_n } is its replacement by a collection ρ = { R_1, R_2, . . ., R_k } of subsets of R such that R = Attr(R_1)∪ Attr(R_2), ∪ . . ∪., Attr(R_k)

There is no requirement that the R_i 's be disjoint. One of the motivations for performing a decomposition is that it may eliminate the problems of anomalies. But decomposition to eliminate anomalies is not the last word in case of database design.

Lossless Joins

If R is a relation scheme decomposed into schemes R_1, R_2, . . ., R_k , and D is a set of dependencies, we say the decomposition is a lossless join decomposition (with respect to D) if for every relation r for R satisfying D:

$$r = \prod_{R1}(r) \bowtie \prod_{R2}(r) \bowtie \ . \ . \ . \bowtie \prod_{Rk}(r)$$

Testing Lossless Joins

Let us take one relation SAIP (supplier_name, address, item, price). Let it has been divided into two – SA and SIP. The fds are – S → A and SI → P. A table may be drawn as follows :

S	A	I	P
a_1	a_2	b_{13}	b_{14}
a_1	b_{22}	a_3	a_4

This table is formed as follows :
Construct a table having columns and rows. Number of columns equals to number of attributes and number of rows equals to number of decomposed relation schemes. Column j corresponds to attribute A_j and row i corresponds to relation scheme R_i . In row i and column j put the symbol a_j if A_j is in R_i . If not, put the symbol b_{ij} there.

Repeatedly consider each of the dependencies X → Y in F, until no more changes can be made to the table. Each time we consider X → Y we look for rows that agree in all the columns for the attributes of X. If we find two such rows, equate the symbols of those rows for the attributes of Y. When we equate two symbols, if one of them is a_j, make the other be a_j . If they are b_{ij} and b_{lj}, make them both b_{ij} or b_{lj} arbitrarily. Since S → A and the two rows agree on S,

we may equate their symbols for A, making b_{22} become a_2. The resulting table is as follows :

S	A	I	P
a_1	a_2	b_{13}	b_{14}
a_1	a_2	a_3	a_4

Since one row has all a's, the join is lossless.

Example :
R = ABCDE, R_1 = AD, R_2 = AB, R_3 = BE, R_4 = CDE, R_5 = AE

Let fds are : A → C, B → C, C → D, DE → C, CE → A

The table :

A	B	C	D	E
a_1	b_{12}	b_{13}	a_4	b_{15}
a_1	a_2	b_{23}	b_{24}	b_{25}
b_{31}	a_2	b_{33}	b_{34}	a_5
b_{41}	b_{42}	a_3	a_4	a_5
a_1	b_{52}	b_{53}	b_{54}	a_5

Apply A → C and B → C to get the following:

A	B	C	D	E
a_1	b_{12}	b_{13}	a_4	b_{15}
a_1	a_2	b_{13}	b_{24}	b_{25}
b_{31}	a_2	b_{13}	b_{34}	a_5
b_{41}	b_{42}	a_3	a_4	a_5
a_1	b_{52}	b_{13}	b_{54}	a_5

Applying fds one by one we get the following :

A	B	C	D	E
a_1	b_{12}	a_3	a_4	b_{15}
a_1	a_2	a_3	a_4	b_{25}
a_1	a_2	a_3	a_4	a_5
a_1	b_{42}	a_3	a_4	a_5
a_1	b_{52}	a_3	a_4	a_5

The above algorithm can be applied for many number of relations. But in case of decomposition into two relations there is a simpler test.

If $\rho = (R_1 , R_2)$ is a decomposition of R , and F is a set of functional dependencies, then ρ has a lossless join w.r.t. F if and only if $(R_1 \cap R_2) \rightarrow (R_1 - R_2)$ or $(R_1 \cap R_2) \rightarrow (R_2 - R_1)$.

Note that these dependencies need not be in the given set F, it is sufficient that they be in F^+.

Example : R = ABC and F = { A → B}. Then the decomposition of R into AB and AC has a lossless join.

Since AB ∩ AC = A, AB - AC = B and A → B holds. But if we decompose R into R_1 = AB and R_2 = BC, $R_1 \cap R_2$ = B, $R_1 - R_2$ = A, $R_2 - R_1$ = C.

Dependency Preserving Decomposition into 3NF

Input : Relation scheme R and set of fds F, which we assume without loss of generality to be a minimal cover.

Output : A dependency preserving decomposition of R such that each relation scheme is in 3NF w.r.t. the projection of F onto that scheme.

Algorithm

```
  i := 0
for each fds X → Y in F do
            begin
                i := i + 1 ;
                Ri := XY;
            end
      if none of the schemes Rj , 1 ≤ j ≤ i
      contains a candidate key for R then
          begin
              i := i + 1;
              Ri := any candidate key for R;
          end
   if Uj=1 Rj ≠ R
     then begin
              Ri+1 := R - Uj = 1 Rj ;
              i:= i + 1;
          end
    return ( R1, R2, . . . , Ri)
```

Lossless Join Decomposition into BCNF

Any relation scheme has a lossless join decomposition into BCNF, and it has a decomposition into 3NF that has a lossless join and is also dependency preserving. However, there may be no decomposition of a relation scheme into BCNF that is dependency preserving.

Algorithm : Lossless join decomposition into BCNF

Input : Relation scheme R and fds F

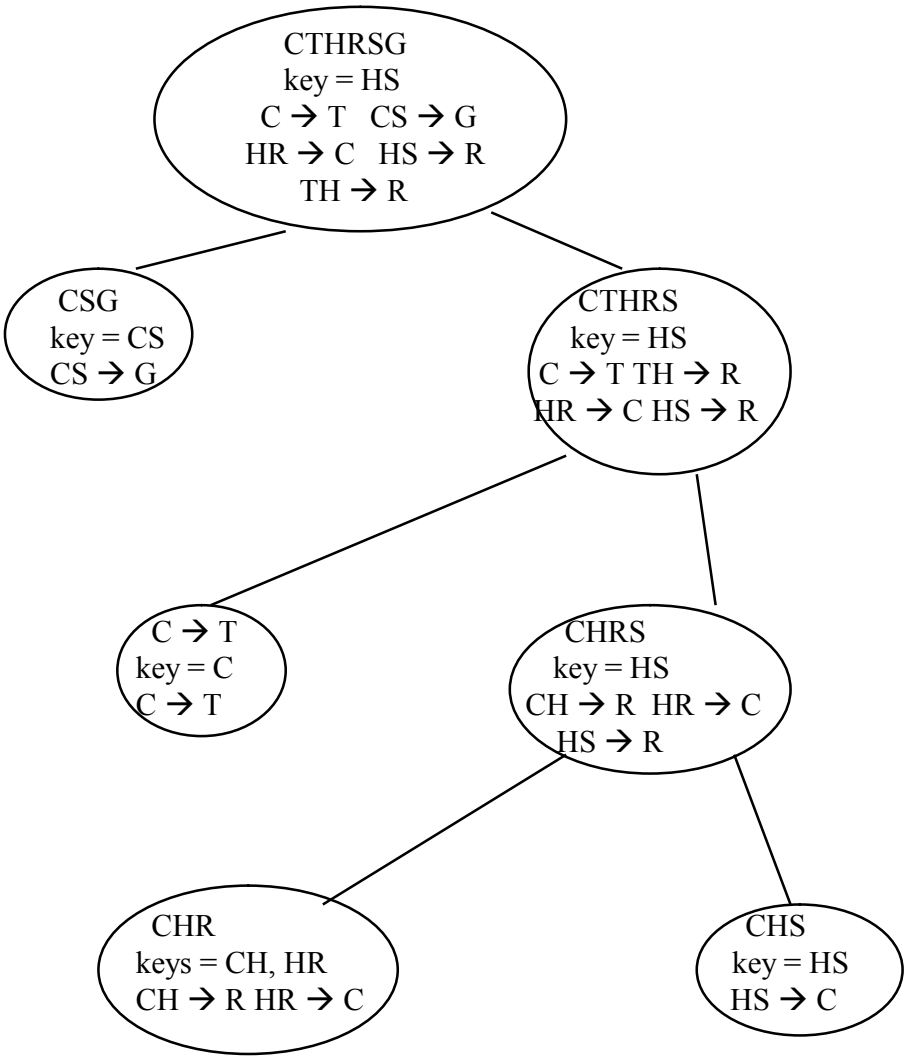
Output : A decomposition of R with a lossless join, such that every relation scheme in the decomposition is in BCNF w.r.t. the projection of F onto that scheme.

Method : We iteratively construct a decomposition ρ for R. At all times ρ will have a lossless join w.r.t. F.

Example Let us consider a relation scheme CTHRSG. The fds are as follows :

C → T Each course has one teacher
HR → C Only one course can meet in a room at one time
HT → R A teacher can be in only one room at one time
CS → G Each student has one grade in each course
HS → R A student can be in only one room at one time

The key is HS. The decomposition is as follows :



Tree of decomposition

The fd TH → R is not preserved here.