Integrity Constraints

The tuples of a RDB represent the current information about some real life object. Generally some constraints or restrictions are imposed on the admissible values of the tuples as per business requirement. These constraints/restrictions essentially determine the semantics of data and are called integrity constraints. It is broadly classified into two categories.

- Integrity constraint on admissible domain values of a tuple :

  These restrict the admissible values of the attributes of the relation to some range and are called domain dependencies. Example : Employee age must be less than or equal to 60, Salary paid to any employee is Gross Salary – Deduction (involve some arithmetic relationship among the different fields of the tuple).

- Integrity constraints based on inter-tuple relationship

  These types of constraints are called data dependencies.
  Example – Relation **Employees**(EmpId, EmpName, DeptID, Grade, Salary, Age Address)
  Restriction that an employee can work in one department only implies that in this relation we cannot find two tuples having same EmpID but differ in DeptID.

Although several types of data dependencies have been reported, they can be broadly classified into two categories,

- Equality generation dependencies – Functional dependency fall under this category.
- Tuple generation dependencies – Multivalued or Join dependencies fall under this category.

Functional Dependency

Few notations :

• Say **R** represents the time-invariant description of a relation. It can also be identified by R (A1, A2, A3 …….. An) where A1..An are attributes. • Set of integrity constraints that must hold for R is denoted by **D** . • An instance (a snapshot of data at a particular time) r of R is called legal instance, if it satisfies D. • Projection of a tuple t of r over X $\subseteq$ R will be donated by t[X] . In practical terms, it can be roughly thought of as picking a subset of all available columns.	Set Theory Symbols $\in$ "is an element of" $\notin$ "is not an element of" $\subset$ "is a *proper* subset of" $\subseteq$ "is a subset of" $\not\subset$ "is not a subset of" $\varnothing$ the empty set; a set with no elements $\cap$ intersection $\cup$ union

• For X, Y ⊆ R, the union X U Y will be denoted by XY	

Before defining functional dependency formally let us examine following example.

Employee number	Employee Name	Salary	City
1	Dana	50000	San Francisco
2	Francis	38000	London
3	Andrew	25000	Tokyo

If we know the value of Employee number, we can obtain Employee Name, city, salary, etc.

So we can say that the city, Employee Name, and salary are functionally depended on Employee number i.e. one attribute determines another attribute in a DBMS system.

Functional Dependency plays a vital role to find the difference between good and bad database design.

Definition of functional dependency

A **functional dependency (FD)** is a **constraint** between two sets of attributes. This constraint is for any two tuples t1 and t2 in r if t1[X] = t2[X] then they have t1[Y] = t2[Y]. This means the value of X component of a tuple uniquely determines the value of component Y.

In other words, at any instance of time the relation cannot contain two tuples that agree in all attributes in the set X yet disagree in one or more attributes in set Y. Equivalently we can say that X identifies Y.

- FD is denoted as X → Y (read as "Y is functionally dependent on X"). The left-hand side of the FD is sometimes called as the determinant and the right-hand side is called dependent.

S#	CITY	P#	QTY
S1	Delhi	P1	100
S1	Delhi	P2	100
S2	Mumbai	P1	200
S2	Mumbai	P2	200
S3	Mumbai	P2	300

For example consider following relation for the shipment, it includes the usual attributes S#, P#, QTY and CITY.

S# → CITY is an FD which satisfies the functional dependency because every tuple of the relation with a given S# value also has the same CITY value.

A functional dependency is a property of the semantics or meaning of the attributes. The database designers need to understand the semantics of the attributes of R to specify the FD that should hold on all relation states r of R. Whenever the semantics of two sets of attributes in R indicate that a functional dependency should hold, we specify the dependency as a constraint.

1. **Fully Functional dependency** : A functional dependency X → Y is full FD if removal of any attribute A from X means that the dependency does not hold any more. This means it must satisfy

- Y is functionally dependent on X
- Y is not functionally dependent on any proper subset of X

2. **Partial Functional dependency :** Y is partially dependent on X , if there is some attribute that can be removed from X and yet the dependency still holds.
   Say for example consider the FD StaffID, Name → BranchID
   BranchID is functionally dependent on a subset of X (StaffID,Name), namely StaffID.

3. "**trivial**" **Functional dependency** (unimportant or insignificant) :

   The dependency of an attribute on a set of attributes is known as trivial functional dependency if the set of attributes includes that attribute. A →B is trivial FD if B is a subset of A.

   The FD A→A & B→B are also trivial as they satisfy by all relations involving attribute A and B.

   For example consider a relation with two columns Student_id and Student_Name.
   {Student_Id, Student_Name} → Student_Id is a trivial functional dependency as Student_Id is a subset of {Student_Id, Student_Name}. Because if we know the values of Student_Id and Student_Name then the value of Student_Id can be uniquely determined.

FD occur naturally in most database. For example, in the Relation Employees(EmpId, EmpName, DeptID, Grade, Salary, Age Address), following Functional dependencies hold.

- EmpID → EmpName   each employee has a unique id
- EmpID → DeptID   an employee can work in one department only
- EmpID, Grade, Age → Salary   employee's salary depends on his age and grade
- EmpID → Age   each employee has unique age
- EmpID → Address   each employee has unique address

EmpID is not functionally dependent on Salary or Age, because more than one employee can have the same salary or can be of same age.

An important consequence of the FD is that if X→Y holds in a relation r, then it also holds in any projection of r that involves XY.

The functional dependencies that hold for a database schema can be **determined only by careful analysis of the meaning of the attributes**. The database designer must have a thorough understanding of the physical system which the conceptual data model is going to represent. After identifying the set of FDs, the DBMS can be asked to enforce these integrity constraints during update operations. While selecting a set of FDs for a relation schema we should try to eliminate any redundant FD from the set.

FDs are used in database systems to help ensure consistency and correctness. **Fewer FDs** mean less **storage space** used and **fewer tests** to make when the database is modified. A smaller set of FDs guarantees faster execution.

Closure of a set of FDs

- In real life, it is impossible to specify all possible functional dependencies for a given situation.
- A **Closure** is a set of all possible FDs that can be derived from a given set of FDs. It is also referred as a **Complete** set of FDs. If F is used to donate the set of FDs for relation R, then a closure of a set of FDs implied by F is denoted by F^+.

- Since any legal instance r of R satisfies $F^+ \supseteq F$ (F is a subset). If F = F^+, F is called a full family of dependencies.

- Even for a relatively small schema, F^+ can be very large. For example consider a relation schema r with three attributes A1, A2, A3 and let F = { A1 → A2, A2 → A3 }. Then F^+ contains **thirty five FDs**.

- So computation of F^+ from F by any procedure is certainly going to be time consuming.
- To determine F^+, we need rules for deriving all functional dependencies that are implied by F. A set of rules (or axioms) used for this purpose was proposed by **Armstrong in 1974**. The rules are stated below.

Armstrong's Axioms

The axioms IR1 to IR3 are independent i.e. none of these axioms can be proved from the other two.

1. Reflexivity: If X is a superset of Y or Y is a subset of X then X → Y.

   Example : SSN,Name → SSN (SSn,Name is superset of SSN)

   Proof : Let Y is a subset of X. Then for any two tuples t1, t2 of r, whenever t1[X] = t2[X], we will always have t1[Y] = t2[Y] (as Y is a subset of X). Hence X → Y

2. Augmentation: If X → Y, then XZ → YZ. Or If Z ⊆W, and X → Y, then XW → YZ

   Example : SSN → Name then SSN,Phone → Name, Phone

   Proof : Suppose that r satisfies X → Y and let Z ⊆W ⊆R. Consider two tuples t1 and t2 of r such that t1[WX] = t2{WX] Since r satisfies X Y, and X values of t1 and t2 are equal, t1[Y] = t2[Y] . Also W values of t1 and t2 being equal, t1[Z] = t2[Z] as Z is a subset of W. Therefore XW YZ holds in r.

3. Transitivity: If X→ Y and Y→ Z, then X→ Z.

   Example : *SSN →Zip* and *Zip → City* then *SSN →City*

   Proof : Suppose that r satisfies X → Y and Y → Z. Then for any two tuples t1 and t2 of r, if t1[X] = t2[X] , by X → Y , t1[Y] = t2{Y]. Again Y→ Z, requires t1[Z] = t2[Z]. Hence r satisfies X → Z

4. Union: If X → Y and X → Z, then X →YZ

Example : SSN →Name and SSN →Zip then SSN →Name,Zip

Proof : Applying IR2 on X → Y, we obtain X Z→ YZ.
Applying IR2 on X → Z, we obtain X → XZ (X union X is X , duplicate will be dropped).
So by IR3 we can conclude X →YZ

5. Decomposition: If X → YZ then X → Y and X →Z.

   Example : SSN→Name,Zip then SSN→Name and SSN→Zip

   Proof: Let X → YZ, By IR1 YZ → Y YZ→ Z. Hence by IR3, we obtain X → Y and X→ Z

6. Pseudo-Transitivity: If X → Y and YW → Z, then XW→ Z.

   Address → Project and Project,Date →Amount then Address,Date → Amount

   Proof : If X → Y, then by IR2 XW → YW. Moreover, if YW → Z then by IR3 XW→ Z

Example

Supposing we are given a relation R {A, B, C, D, E, F} with a set of FDs as shown below: A → BC, B→ E , CD→ EF . Show that the FD AD → F holds for R and is a member of the closure.

1. A → BC & CD→ EF	{Given}
2. A → C & A→ B	{Decomposition of (1)}
3. AD → CD	{Augmentation of (2) by adding D}
4. AD→ CD & CD→ EF so AD→ EF	{Transitivity of (3) and (1)}
5. AD→ E **AD → F**	{Decomposition of (4)}

Based on the set of above inference rules, a simple way to detect and remove redundant FDs would be as follows:

1. Begin with the given set of FDs F.
2. Remove an FD, f , and create a set of FDs F ‘ = F – {f}
3. Test whether f can be derived from FDs in F’ using the set of inference rules
4. If f can be inferred from F’, it is redundant and hence set F = F’
5. Repeat steps 2 to 4 for all FDs in F

Closure of a set of Attributes

We will now define closure of a set of attributes with respect to a given set of FDs.

Given a set of FDs F of a relation schema R and let X be a set of attributes ($X \subseteq R$). The closure of X with respect to F, denoted by **X^+ (F)** or simply **X^+**, is the set of all attributes A ε R, such that $X \rightarrow A$ can be inferred from F using the inference axioms IR1 to IR6.

Thus X^+ contains all attributes of R which are functionally dependent on X. By reflexivity rule the closure of X always contains X.

Example : Consider the set of FDs { $A \rightarrow BC$, $AC \rightarrow D$, $D \rightarrow B$, $AB \rightarrow D$ }

We can inferred using axioms $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow AB$ (as A union A is A),
$A \rightarrow D$ by transitivity ($A \rightarrow AB$, $AB \rightarrow D$).

The closure **A^+ = { A, B, C, D } and D^+ = {B,D}**

Thus A is the Key of the relation because all other attributes depends of A (all attributes appears in the closure of A)

Lemma : An FD $X \rightarrow Y$ can be inferred from a given set of FDs F using the inference axioms IR1 .. IR6, if and only if $Y \subseteq X^+$

In view of the above lemma, following algorithm can be used to detect whether an FD $X \rightarrow Y$ can be derived from a given set of FDs (F). The algorithm first computes X^+ and then checks whether $Y \subseteq X^+$

Membership Algorithm

Algorithm : Member
Input : A set of FDs and another FD $X \rightarrow Y$ (to be derived from the set)
Output : True, if $X \rightarrow Y$ can be inferred from F, False otherwise

Begin

1. XPLUS := X /* Initialization step. By reflexivity rule $X \rightarrow X$ */
2. Look at FDs in F to see, if there exists an FD $Z \rightarrow V$ ε F such that $Z \subseteq$ XPLUS and $V \not\subseteq$ XPLUS, then **set XPLUS := XPLUS U V** /* By union and transitivity rule */
3. Repeat step 2 every time XPLUS is changed until no more attributes can be added to XPLUS /* When finally exits from this step XPLUS is the closure of X with respect to F */
4. If Y $\subseteq$ XPLUS, then return True else return False

End;

To illustrate how the algorithm works, we shall use the algorithm to compute $(AG)^+$.

Suppose we are given a relation schema R = (A, B, C, G, H, I) and
the set of FDs {A → B, A → C, CG →H, CG →I, B → H}.

result = AG;	initialized value
result = ABG;	result := result U B because A→ B exists and A is subset of result.
result = ABCG;	because A → C
result = ABCGH;	because CG →H
result = ABCGHI ;	because CG → I

Cover for Functional Dependencies

We can visualize the closure F^+ as a complete set of information regarding the integrity constraints that can be extracted from the given set of dependencies with the help of Armstrong's axioms. F^+ may contain large number of dependencies and most of them may be redundant. So, for the design point of view we are more interested in finding minimal non-redundant set of FDs than F^+.

Let us first define the cover.

Let F and G are two sets of FDs of a relation R. We say F is a cover of G, if $F^+ = G^+$ (i.e. closure of these two set of dependencies are equal)

If F covers G, we can say F is equivalent to G.

Example : Consider a Relation **Publication** (Publisher, Title, Year, Subject, NoPages, Author, Price). Following dependencies hold

f1: Publisher, Title, Year → Subject, NoPages
f2: Publisher, Title, Year → Author
f3: Subject, NoPAges → Price
f4: Publisher, Title, Year → Price

The set of FDs F = {f1, f2, f3, f4 } **covers** the set G = {f1, f2, f3}. f4 itself can be inferred from f1 and f3 by transitivity rule. Hence for maintaining data integrity, the DBMS can be asked to enforce f1, f2, f3 only. In this process f4 will be automatically satisfied.

Minimal Cover

While finding a cover of a given set of FDs, say F, it is useful to find the minimal cover F' of F. F' is said to be a minimal cover of F if :

1. every right hand side of each dependency in F' has only one attribute
2. for any f ε F' , F' – {f} is not equivalent to F (not a cover of F) . This means we cannot remove any dependency from F' and still have set of dependency that is equivalent to F.

3. for no $X \rightarrow A$ in F' and proper subset Z of X,

   ((F' - {$X \rightarrow A$}) U {$Z \rightarrow A$} equivalent to F (cover of F). We cannot replace any dependency $X \rightarrow A$ in F' with a dependency $Z \rightarrow A$ where Z is a proper subset of X and still have a set of dependency that is equivalent to F

d) implies that no dependency in F' is redundant.
e) guarantees that no attributes on any left side is redundant i.e. each FD is a fully functional dependency.

Moreover, using decomposition axiom we can always reduce right side of each FD in F' to have only one attribute. Thus each FD in F is not only non-redundant but does not contain any redundant attribute neither in the left nor in the right side. Such FDs are called reduced FDs.

The minimal set may be considered a standard or canonical form of FDs with no redundancies that is equivalent to F. Unfortunately however the minimal cover is not unique.

The membership algorithm described earlier can be used to find a minimal cover of a set of FDs.

Normalization

While designing a Relational data model, database designed must be aware of all the integrity constraints that need to be satisfied and ensure that database consistency is preserved following any update operations. If the relational schemes are not properly chosen anomalies can occur after database tuple operation. For Example consider the schema **Student**(Name, Address, Subject, Grade)

There are several problems associated with this schema.

- **Redundancy**: The student's address is repeated for each subject he is registered for.
- **Update Anomaly** : We may update the address in one tuple, while leaving it unchanged in another. Thus we would not have a unique address for each student.
- **Insertion Anomaly** : It is not possible to record the address of a student, unless he has registered for at least one subject. Also, we may insert a different address, when a new tuple indication his registration in a new course is added.
- **Deletion Anomaly** : If a student drops all the subjects in which he is registered, student's address will be lost.

Database design goal

Represent the user data by relations that do not create anomalies following tuple add, delete, or update operations. This can only be achieved by a careful analysis of the integrity constraints, especially the data dependencies, of the database.

Designing the relations starts with a number of groupings of attributes into relations that exist together naturally. This is mainly based on understanding of meaning of data by the designer and some informal guidelines. However, we still need some **formal measure** to ensure our design goal. Why one grouping of attributes into a relation schema may be better than another?

Normalization is that formal measure.

What is Normalization?

Normalization is a systematic way of ensuring that a database structure is suitable for general-purpose querying and free of certain undesirable characteristics—insertion, update, and deletion anomalies—that could lead to a loss of data integrity.

In this process we successively decompose the tables to reach

- tables with fewer columns with proper relationship which will make data retrieval and insert, update and delete operations more efficient and
- eliminate data redundancy and reduce the chance of going to an inconsistent state after any operation.

Normalization helps to achieve followings:

- reduce the amount of space a database consumes by eliminating data redundancy
- make the relational model more informative to users
- reduce the chance of going to an inconsistent state after any operation

The Normal Forms

The database community has developed a series of guidelines for ensuring that databases are normalized. These are referred to as normal forms and are numbered from one (the lowest form of normalization, referred to as first normal form or 1NF) through seven (seven normal form or 7NF). In practical applications, you'll often see 1NF, 2NF, and 3NF along with the occasional 4NF. Other normal forms are rarely used.	There are seven normal forms exist as of today: • First Normal Form • Second Normal Form • Third Normal Form • Boyce-Codd Normal Form • Fourth Normal Form • Fifth Normal Form • Sixth or Domain-key Normal form

First normal form (1NF or Minimal Form)

A relational database table that adheres to 1NF is one that meets a certain minimum set of criteria. These criteria are basically concerned with ensuring that the table is a faithful representation of a relation and that it is **free of repeating groups**.

According to Date's definition of 1NF, a table is in 1NF if and only if it is "isomorphic to some relation", which means, specifically, that it satisfies the following conditions:

1. There's no top-to-bottom ordering to the rows and left-to-right ordering to the columns.
2. There are no duplicate rows.
3. Every row-and-column intersection contains exactly one value from the applicable domain or null value.

   Most people think this condition (3) as the defining feature of 1NF. It is primarily concerned with repeating groups. This condition indicates that Column values should be atomic, scalar or should be holding single value. No repetition of information or values in multiple columns.

4. All columns are regular [i.e. rows have no hidden components such as row IDs, object IDs, or hidden timestamps].

Violation of any of these conditions would mean that the table is not strictly relational, and therefore that it is not in 1NF.

Examples of tables (or views) that would not meet this definition of 1NF are:

- A table that lacks a unique key. Such a table would be able to accommodate duplicate rows, in violation of condition 2.
- A view whose definition mandates that results be returned in a particular order, so that the row-ordering is an intrinsic and meaningful aspect of the view. This violates condition 1.

Repeating groups

The following scenario illustrates how a database design might incorporate repeating groups, in violation of 1NF.

Repeating groups within columns – First solution

Customer ID	First Name	Surname	Telephone Numbers
123	Bimal	Saha	555-861-2025 456
456	Kapil	Khanna	555-403-1659, 555-776-4100 789
789	Kabita	Roy	555-808-9633

Suppose a designer wishes to record the names and telephone numbers of customers. He defines a customer table as shown

Here Telephone Number column is not atomic or doesn't have scalar value i.e. it has having more than one value. So it is not 1NF

A query such as "Which pairs of customers share a telephone number?" is more difficult to formulate.

Repeating groups across columns – 2nd Solution [Three separate columns for telephone nos.]

Customer ID	First Name	Surname	Tele No1	Tele No2	Tele No3
123	Bimal	Saha	555-861-2025 456		
456	Kapil	Khanna	555-403-1659	555-776-4100 789	
789	Kabita	Roy	555-808-9633		

Tele No1, Tele No2, and Tele No3 share exactly the same domain and exactly the same meaning; the splitting of Telephone Number into three headings is artificial and causes logical problems. These problems include:

- Difficulty in querying the table. Answering such questions as "Which customers have telephone number X?"
- Inability to enforce uniqueness of Customer-to-Telephone Number links through the RDBMS. Customer 789 might mistakenly be given a Tele No2 value that is exactly the same as her Tele No1 value.
- Restriction of the number of telephone numbers per customer to three. If a customer with four telephone numbers comes along, we are constrained to record only three and leave the fourth unrecorded. This means that the database design is imposing constraints on the business process.

To make it 1NF

- We'll first break (decompose)our single table into two.
- Each table should have information about only one entity.

Customer Table

Customer ID	First Name	Surname
123	Bimal	Saha
456	Kapil	Khanna
789	Kabita	Roy

Telephone Table

Customer ID	Tele No
123	555-861-2025 456
456	555-403-1659
456	555-776-4100 789
789	555-808-9633

Repeating groups of telephone numbers do not occur in this design. Instead, each Customer-to-Telephone Number link appears on its own record.

It is worth noting that this design meets the additional requirements for second and third normal form.

Atomicity

Some definitions of 1NF, most notably that of Edgar F. Codd, make reference to the concept of atomicity. Date suggests that "the notion of atomicity has no absolute meaning": a value may be considered atomic for some purposes, but may be considered an combination of more basic elements for other purposes (example date as combination of dd mm and yyy). If this position is accepted, 1NF cannot be defined with reference to atomicity.

Second Normal Form

Any table that is in second normal form (2NF) or higher is, by definition, also in 1NF (each normal form has more stringent criteria than its predecessor). On the other hand, a table that is in 1NF may or may not be in 2NF; if it is in 2NF, it may or may not be in 3NF, and so on.

To understand the 2NF consider following table

Customer Table			
Customer id	Email	First Name	Surname
108	kapil.dev@google.com	Kapil	Dev
252	sudip@yahoo.co.in	Sudip	Sinha
252	sudip@google.com	Sudip	Sinha
360	babita@yahoo.in	Babita	Kulkarni
360	babita@google.com	Babita	Kulkarni

We are storing more than one email of customer in this table so key is {Customer ID, Email}

If Babita changes her surname by marriage, the change must be applied to two rows. If the change is only applied to one row, we will get inconsistent result while querying. 2NF addresses this problem.

Specifically: a 1NF table is in 2NF if and only if none of its non-prime attributes are functionally dependent on a part (proper subset) of a candidate key. (A non-prime attribute is one that does not belong to any candidate key.)

Here first name and surname are non-prime attributes. They are functionally dependent on part of primary key i.e. customer id or email.

So a relation is in 2NF if it is in 1NF and every non-prime attribute of the relation is dependent on the whole of every candidate key.

Note that when a 1NF table has no composite candidate keys (candidate keys consisting of more than one attribute), the table is automatically in 2NF.

Example 1 :

Gadgets	Supplier	Cost	Supplier Address
Headphone	Abaci	123$	New York
MP5 Player	Sagas	250$	California
Headphone	Mayas	100$	London

In this table Gadgets +SUPPLIER together form a composite primary key. Let's check for dependency of each non-key column.

Start with cost column

- If I know gadget can I know the cost? - No same gadget is provided my different supplier at different rate.
- If I know supplier can I know about the cost? - No because same supplier can provide me with different gadgets.
- If I know both gadget and supplier can I know cost? Yes than we can.
- **So cost is fully dependent (functionally dependent) on our composite primary key (Gadgets+Supplier)**

Let's consider another non-key column Supplier Address.

- If I know gadget will I come to know about supplier address? - Obviously no.
- If I know who the supplier is can I have it address? - Yes.
- So here supplier is not completely dependent on (partial dependent) composite primary key (Gadgets+Supplier).

This table is surely not in Second Normal Form. To make it 2NF we have to decompose the table.

Example 2 :

Employees' Skills		
Employee	Skill	Current Work Location
Jones	Typing	114 Main Street
Jones	Shorthand	114 Main Street
Jones	Whittling	114 Main Street
Bravo	Light Cleaning	73 Industrial Way
Ellis	Alchemy	73 Industrial Way
Ellis	Flying	73 Industrial Way
Harrison	Light Cleaning	73 Industrial Way

Here {Employee, Skill} is a candidate key for the table. This is because a given Employee might have more than one skill. Similarly more than one employee have same skill.

The remaining attribute, Current Work Location, is dependent on only part of the candidate key, namely Employee. Therefore the table is not in 2NF.

Note the redundancy in Current Work Locations : we are told three times that Jones works at 114 Main Street, and twice that Ellis works at 73 Industrial Way. This redundancy makes the table vulnerable to update anomalies. So the query "What is Jones' current work location?" may give inconsistent result.

Employees	
Employee	**Current Work Location**
Jones	114 Main Street
Bravo	73 Industrial Way
Ellis	73 Industrial Way
Harrison	73 Industrial Way

Employees' Skills	
Employee	**Skill**
Jones	Typing
Jones	Shorthand
Jones	Whittling
Bravo	Light Cleaning
Ellis	Alchemy
Ellis	Flying
Harrison	Light Cleaning

So we have to apply decomposition to make it 2NF.

- "Employees" table with key {Employee}
- "Employees' Skills" table with key {Employee, Skill}.

Neither of these tables can suffer from update anomalies and redundancy.

However not all 2NF tables are free from update anomalies, see the following example of a 2NF table which suffers from update anomalies.

Tournament Winners			
Tournament	**Year**	**Winner**	**Winner Date of Birth**
Des Moines Masters	1998	Chip Masterson	14 March 1977
Indiana Invitational	1998	Al Fredrickson	21 July 1975
Cleveland Open	1999	Bob Albertson	28 September 1968
Des Moines Masters	1999	Al Fredrickson	21 July 1975
Indiana Invitational	1999	Chip Masterson	14 March 1977

Here Winner and Winner Date of Birth are determined by the whole key {Tournament / Year} and not part of it. So it satisfies 2NF.

But **redundancy** (particular Winner / Winner Date of Birth combinations are shown on multiple records) leads to an update anomaly: if updates are not carried out consistently, a particular winner could be shown as having two different dates of birth.

Winner Date of Birth actually depends on Winner, which in turn depends on the key Tournament / Year. So a **transitive dependency** exist which is the cause of this anomaly.

Third normal form

Codd's definition states that a table is in 3NF if and only if both of the following conditions hold:

- The relation R (table) is in second normal form (2NF)
- Every non-key attribute of R is **non-transitively dependent** (i.e. directly dependent) on the primary key of R. This means no **nonprime attribute** (not part of candidate key) is functionally dependent on **other nonprime attributes**.

To understand the third normal form, we need to understand transitive dependence which is based on one of Armstrong's axioms. Let A, B and C be three attributes of a relation R such that

A → B and B → C. From these FDs, we may derive A → C. This dependence **A→ C is transitive.**

Existence of transitive dependence is an **indication** that the relation has information about more than one thing and should therefore be decomposed.

For example consider the relation **subject** (cno, cname, instructor, office)

Assume that cname is not unique and therefore **cno** is the only candidate key. The following functional dependencies exist : cno → cname, cno → instructor, instructor → office

We can derive cno → office from the above functional dependencies and therefore the above relation is in 2NF (all non-prime attribute depends on prime attribute). But the relation is not in 3NF since office is not directly dependent on cno (it is a transitive dependency) . This transitive dependence is an indication that the relation has information about more than one thing (viz. course and instructor).

A 3NF definition that is equivalent to Codd's, but expressed differently, was given by Carlo Zaniolo in 1982.

This definition states that a table is in 3NF if and only if, for each of its functional dependencies **X → A**, at least one of the following conditions holds:

- X contains A (that is, X → A is trivial functional dependency), or
- X is a super key, or
- A is a prime attribute (i.e., A is contained within a candidate key)

An attribute or a combination of attribute that is used to identify the records **uniquely** is known as Super Key. A table can have many Super Keys.

Zaniolo's definition gives a clear sense of the difference between 3NF and the more stringent Boyce-Codd normal form (BCNF). BCNF simply eliminates the third alternative ("A is a prime attribute").

In the above example in FD Winner → Winner Date of Birth, winner is not a super key. Neither Winner Date of Birth is part of candidate key nor is a trivial dependency. So it is not satisfying any of the above condition and the relation is not 3NF.

Transforming 2NF to 3NF

Example -1

Tournament Winners

Tournament	Year	Winner	Winner Date of Birth
Indiana Invitational	1998	Al Fredrickson	21 July 1975
Cleveland Open	1999	Bob Albertson	28 September 1968
Des Moines Masters	1999	Al Fredrickson	21 July 1975
Indiana Invitational	1999	Chip Masterson	14 March 1977

This table is in 2NF with a composite key {Tournament, Year}.

Not in 3NF as discussed earlier (Winner → Winner Date of Birth transitive dependency exists.

{Tournament, Year} → Winner   Winner → Winner Date of Birth
So, {Tournament, Year} → Winner Date of Birth (transitive)

The fact that Winner Date of Birth is functionally dependent on Winner makes the table vulnerable to logical inconsistencies, as there is nothing to stop the same person from being shown with different dates of birth on different records. In order to express the same facts without violating 3NF, it is necessary to split the table into two:

Tournament Winners		
Tournament	**Year**	**Winner**
Indiana Invitational	1998	Al Fredrickson
Cleveland Open	1999	Bob Albertson
Des Moines Masters	1999	Al Fredrickson
Indiana Invitational	1999	Chip Masterson

Player Dates of Birth	
Player	**Date of Birth**
Chip Masterson	14 March 1977
Al Fredrickson	21 July 1975
Bob Albertson	28 September 1968

Update anomalies cannot occur in these tables, which are both in 3NF.

Example -2

Consider the relation Order (Order#, Part, Supplier, UnitPrice, QtyOrdered) with FDs
Order# → Part , Supplier, QtyOrdered and Supplier, Part → UnitPrice) Here Order# is key.

By Amstrong's axioms we can Order# → Part , Order→ Supplier, and Order →QtyOrdered.
Again Order# →Part ,Supplier and Supplier, Part → Unit Price, so Order# → UnitPrice.

So all nonprime attributes depends on key, but there is transitive dependency between UnitPrice and Order#. Hence it is not in 3NF.

We cannot store UnitPrice information of any part supplied by any Supplier unless an order has been placed for that part. So we need to decompose to make it 3NF.

Order (Order#, Part, Supplier, QtyOrdered) and **Price Master** (Part, Supplier, UnitPrice)

Note :

- A **super key** is any combination of columns that uniquely identifies a row in a table.
- A **candidate key** is a super key which cannot have any columns removed from it without losing the unique identification property. This property is sometimes known as minimality or (better) irreducibility.
- A super key ≠ a primary key **in general**. The primary key is simply a candidate key chosen to be the main key.

Boyce-Codd normal form

A relational R is considered to be in **Boyce–Codd normal form (BCNF)** if it satisfies the following two conditions:

1. It should be in the **Third Normal Form**.
2. For every FD X → Y, one of the following conditions holds true:
   - X → Y is a trivial functional dependency (i.e., Y is a subset of X)
   - X is a **superkey** for schema R

Informally the Boyce-Codd normal form is expressed as *"Each attribute must represent a fact about the key, the whole key, and nothing but the key."*

In simple words, it also means, that for a dependency X → Y,

- X cannot be a **non-prime attribute**, if Y is a **prime attribute** (no prime attribute can depend on nonprime attribute).

Example

StudentId	Subject	Professor
HIT2003	Angular	P.Angular1
HIT2003	Object Oriented Technology	P.OOT
HIT2901	Angular	P.Angular2
HIT2902	Database	P.DBMS
HIT2904	Angular	P.Angular1

- One student can enroll for multiple subjects. For example, student with **StudentId** HIT2003, has opted for subjects – Angular & OOT
- For each subject, a professor is assigned to the student.
- And, there can be multiple professors teaching one subject like we have for Angular.

What do you think should be the **Primary Key**?

Note: no single attribute is a candidate key

Primary key can be StudentI, Subject or StudentId, Professor. With the help of these keys we can find all the columns of the table.

Here we assume **StudentId, Subject** together form the primary key.

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. (**Professor → Subject**)

- This table satisfies the 1NF as no repeating field 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 (non prime attribute Professor fully depends on prime attributes StudentId and Subject)
- And, there is no Transitive Dependency, hence the table also satisfies the 3rd Normal Form.

But this table is not in Boyce-Codd Normal Form.

Subject which is a part of composite candidate key is determined by non-key attribute Professor of the same table, which is against the rule. Operation on above table can generate following anomalies too.

ANOMALIES

- **Deleting** student deletes professor info
- **Insert** a new professor – need a student
- **Update** – inconsistencies. If we update subject for any student, his professor info also needs to be changed, else it will lead to inconsistency.

Why this table is not in BCNF?

StudentId and Subject form primary key means subject column is a prime attribute. And there is one more dependency, **Professor → Subject**.

is a non-prime attribute

a prime attribute

This type of FD is not allowed by BCNF.

How to satisfy BCNF?

To make this relation(table) satisfy BCNF, we will decompose this table into two tables, **student** table and **professor** table. Below we have the structure for both the tables.

StudentId	ProfessorId
HIT2003	P01
HIT2003	P02
HIT2901	P03
HIT2902	P04
HIT2904	P01

Student Table

ProfessorId	Subject	Professor
P01	Angular	P.Angular1
P02	Object Oriented Technology	P.OOT
P03	Angular	P.Angular2
P04	Database	P.DBMS

Professor Table

And now, this relation satisfy Boyce-Codd Normal Form.

Example : Let's assume there is a company where employees work in more than one department.

- A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
- For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

EMPLOYEE table:

EMP_ID	EMP_COUNTRY	EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
264	India	Designing	D394	283
264	India	Testing	D394	300
364	UK	Stores	D283	232
364	UK	Developing	D283	549

Candidate key: {EMP-ID, EMP-DEPT}

In the above table Functional dependencies are as follows:

EMP_ID → EMP_COUNTRY EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

The table is not in BCNF because neither EMP_DEPT nor EMP_ID alone are keys. So left hand side of above dependencies is not superkey.

To convert the given table into BCNF, we decompose it into three tables:

EMP_COUNTRY table:

EMP_ID	EMP_COUNTRY
264	India
264	India

EMP_DEPT table:

EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
Designing	D394	283
Testing	D394	300
Stores	D283	232
Developing	D283	549

EMP_DEPT_MAPPING table:

EMP_ID	EMP_DEPT
D394	283
D394	300
D283	232
D283	549

Functional dependencies:

1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate keys:

For the first table: EMP_ID
For the second table: EMP_DEPT
For the third table: {EMP_ID, EMP_DEPT}

Now, this is in BCNF because left side part of both the functional dependencies is a key.

Example

Genre means a style or category of art, music, or literature.

Let's take a look at this table, with some typical data. The table is not in BCNF.

Author	Nationality	Book title	Genre	Number of pages
William Shakespeare	English	The Comedy of Errors	Comedy	100
Markus Winand	Austrian	SQL Performance Explained	Textbook	200
Jeffrey Ullman	American	A First Course in Database Systems	Textbook	500
Jennifer Widom	American	A First Course in Database Systems	Textbook	500

The nontrivial functional dependencies in the table are:

key is {**author, book title**}.

author → nationality   book title → genre, number of pages

The same data can be stored in a BCNF schema. However, this time we would need three tables.

Let's see the FD: **book title → genre, number of pages**

This FD is violating the BCNF rules. We split our relation into two relations as shown below.

Book title	**Genre**	**Number of pages**
The Comedy of Errors	Comedy	100
SQL Performance Explained	Textbook	200
A First Course in Database Systems	Textbook	500

One table with all attributes of FD (book title, genre, number of pages)

Author	**Nationality**	**Book title**
William Shakespeare	English	The Comedy of Errors
Markus Winand	Austrian	SQL Performance Explained
Jeffrey Ullman	American	A First Course in Database Systems
Jennifer Widom	American	A First Course in Database Systems

Another table with left side attribute of FD (book title) and remaining attributes (Author, Nationality)

The (book title, genre, number of pages) table is in BCNF. But (book title, author, nationality) isn't. We have the dependency author → nationality

We have to decompose the table one more time. This time we decompose into:

1. columns forming the functional dependency: (author, nationality)
2. the remaining columns: (author, book title)

This time every table is in BCNF.

Author	**Nationality**
William Shakespeare	English
Markus Winand	Austrian
Jeffrey Ullman	American
Jennifer Widom	American

Book title	**Genre**	**Number of pages**
The Comedy of Errors	Comedy	100
SQL Performance Explained	Textbook	200
A First Course in Database Systems	Textbook	500

Author	**Book title**
William Shakespeare	The Comedy of Errors
Markus Winand	SQL Performance Explained
Jeffrey Ullman	A First Course in Database Systems
Jennifer Widom	A First Course in Database Systems

It satisfies all above functional dependencies without violating the BCNF rules, so the schema is in Boyce-Codd normal form.

1st Table key - {author}.
2nd Table key {book title}.
3rd Table key {author, book title}.

How Do You Decompose Your Schema into Boyce-Codd Normal Form?

To go from non-BCNF normal form to BCNF, you must decompose your table using these two steps.

1. Find a nontrivial functional dependency $X \rightarrow Y$ which violates the BCNF condition (where the X is not a superkey)
2. Split your table in two tables:
    - one with attributes XY (all attributes from the dependency),
    - one with X attributes together with the remaining attributes from the original relation

Then you keep **repeating** the decomposition process until all of your tables are in BCNF. After sufficient iterations you have a set of tables, each in BCNF, such that the original relation can be reconstructed.

Step by Step Process towards 3NF

In the **First** Relation Supplier and their shipment quantity for different parts are shown. We assumed the supplier's status is determined by the corresponding location (City). The primary Key of First is (S#, P#). Here the relation is in 1NF because all attributes are non-repeating value but not in 2 NF because Status and City are not fully functional dependent of the primary key (S#, P#). City and Status are also not mutually independent. So this relation suffers from following anomalies :

1. Inserting – No new supplier can be added until it supplies at least one part, because the primary key is (S#,P#)
2. Deleting – If supplier supplies only one part, and we delete that tuple, then we destroy not only the shipment information but also the supplier's location information (tuple of S3).
3. Updating – City and Status of a supplier appears many times in First relation. So possibility of inconsistency during update may arise

First

S#	Status	City	P#	Qty
S1	20	Kolkata	P1	100
S1	20	Kolkata	P2	200
S1	20	Kolkata	P3	400
S1	20	Kolkata	P4	300
S1	20	Kolkata	P5	100
S1	20	Kolkata	P6	500
S2	10	Delhi	P1	300
S2	10	Delhi	P2	400
S3	10	Delhi	P2	200
S4	20	Kolkata	P2	200
S4	20	Kolkata	P4	300
S4	20	Kolkata	P5	400

FDs in this relation :
S#,P# → Qty, S# → Status

S# → City , City → Status

It is not 2 NF.

Second

S#	Status	City
S1	20	Kolkata
S2	10	Delhi
S3	10	Delhi
S4	20	Kolkata
S5	30	Mumbai

FDs in this relation :

S# → City City → Status
S# → Status (by transitivity)

Primary key is S#, all non prime attributes fully depend on the primary key. So it is in 2 NF

SP

S#	P#	Qty
S1	P1	100
S1	P2	200
S1	P3	400
S1	P4	300
S1	P5	100
S1	P6	500
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

FDs in this relation :

S#, P# → Qty

It is 2 NF as well as 3 NF

The solution of above problem is to decompose the First into two relations **Second** and **SP** as show above. We can enter a new supplier S5, although he does not supply any part. Now both the relations are in 2 NF. Original relation (First) can always be recovered by taking the natural join of these projections (Second & SP), so no information is lost during this process (nonloss decomposition) . In other words, the process is reversible.

So any information that can be derived from the original relation (First) can also be derived from the decomposed relations (Second, SP). After natural join of Second and SP on S#, we will get all the tuples of First. The new supplier S5 will not appear after join. But reverse is not true. We cannot reproduce S5 tuples by decomposing the First. In this sense the new decomposed structures are slightly more faithful.

The relation SP is now trouble free and in 3 NF. But the relation **Second** is not 3NF . Here Status depends on S# (primary key) via City (transitivity). It will cause following anomalies:

1. Inserting – Not possible to add new city with status, until any supplier is there in that city because S# is the primary key.
2. Deleting – If we delete a supplier, the information of city's status may be lost if it is the only supplier of that city (for example S5 tuple)
3. Updating – City and Status appears many times. So possibility of inconsistency during update may arise.

SC

S#	City
S1	Kolkata
S2	Delhi
S3	Delhi
S4	Kolkata
S5	Mumbai

FD : S# → City

CS

City	Status
Kolkata	20
Delhi	10
Mumbai	30

FD : City → Status

To overcome above anomalies we can decompose the **Second** into two relations **SC** and **CS** as shown. The process is reversible, once again, since Second is the join of SC and CS over City. Now both the relations are in 3 NF.

It is not possible to just to look at the tabulation of a given relation at a given time and to say whether or not that relation is 3 NF – it is necessary to know the meaning of the data, i.e., the dependencies involved., before that a judgment can be made.

In particular, the DBMS cannot ensure that a relation is maintained in 3 NF (or any other given form, except 1 NF) without being informed of all relevant dependencies. For a relation in 3 NF, however, all that is needed to inform the DBMS of those dependencies is an indication of attribute(s) constituting the primary key. The DBMS will then know that all other attributes are functionally dependent on this attribute(s), and will be able to enforce this constraint.

- Relation **First** contains three determinants: S#, City and (S#,P#). Only the (S#,P#) is the candidate key. Hence First is not BCNF
- Relation **Second** contains determinants: S#, City . But City is not the candidate key. Hence Second is not BCNF.
- Relation **SP**, **SC** and **CS** are each BCNF, because in case primary key is the only determinant.

Multivalued Dependencies

- Functional dependencies rule out certain tuples from appearing in a relation. If A → B, then we cannot have two tuples with the same A value but different B values.
- Multivalued dependencies on the other hand require that tuples of a certain form be present in the relation. Hence Multivalued dependencies (MVD) are classified under **tuple generating type** of dependencies.
- Intuitively, a multivalued dependency **X →→ Y** read as "there is a multivalued dependency of Y on X" or "X multidetermines Y",
- A functional dependency is a special case of multivalued dependency. In a functional dependency X → Y, every x determines exactly one y, never more than one.

Consider the following relation that represents an entity employee that has one mutlivalued attribute proj: **emp (e#, dept, salary, proj)**

- Here e# → dept implies only one dept value for each value of e#.
- But not all information in a database is single-valued, example, proj in an employee relation may be the list of all projects that the employee is currently working on. Although e# determines the list of all projects that an employee is working on, **e# → proj** is not a **functional dependency**.

The fourth and fifth normal forms deal with multivalued dependencies. Before discussing the 4NF and 5NF we discuss the following example to illustrate the concept of multivalued dependency.

programmer (emp_name, qualifications, languages)

Two multivalued attributes qualifications and languages exist. There are no functional dependencies.

emp_name	qualifications	languages
SMITH	B.Sc	FORTRAN
SMITH	B.Sc	COBOL
SMITH	B.Sc	PASCAL
SMITH	Dip.CS	FORTRAN
SMITH	Dip.CS	COBOL
SMITH	Dip.CS	PASCAL

emp_name	qualifications	languages
SMITH	B.Sc	NULL
SMITH	Dip.CS	NULL
SMITH	NULL	FORTRAN
SMITH	NULL	COBOL
SMITH	NULL	PASCAL

emp_name	qualifications	languages
SMITH	B.Sc	FORTRAN
SMITH	Dip.CS	COBOL
SMITH	NULL	PASCAL

(1)

emp_name	qualifications
SMITH	B.Sc
SMITH	Dip.CS

emp_name	languages
SMITH	FORTRAN
SMITH	COBOL
SMITH	PASCAL

(2)

All these variations have some disadvantages such as repeating information and anomalies. If there is no repetition (as shown in third option in fig(1)), the role of NULL values in the above relations is confusing. Also the key is (emp name, qualifications language) and existential integrity requires that no NULLs be specified.

- The attributes qualifications and languages are assumed independent of each other.

- The above relation is therefore in 3NF (even in BCNF) but it still has some disadvantages. Suppose a programmer has several qualifications (B.Sc, Dip. Comp. Sc, etc) and is proficient in several programming languages. We can represent it several ways, three of them are shown below in fig(1)

- The problem in the relation may be overcome by decomposing it. Consider **qualifications** and **languages** separate entities as shown in fig (2). There are two relationships exist (one between employees and qualifications and the other between employees and programming languages).

- Both the above relationships are many-to-many i.e. one programmer could have several qualifications and may know several programming languages. Also one qualification may be obtained by several programmers and one programming language may be known to many programmers.

The basis of the above decomposition is the concept of multivalued dependency (MVD). Functional dependency A → B relates one value of A to one value of B while multivalued dependency A → → B defines a relationship in which a set of values of attribute B are determined by a single value of A.

Fourth and Fifth Normal Forms

Fourth and fifth normal forms deal with multi-valued facts. The multi-valued fact may correspond to a many-to-many relationship, as with employees and skills, or to a many-to-one relationship, as with the children of an employee (assuming only one parent is an employee). By "many-to-many" we mean that an employee may have several skills, and a skill may belong to several employees.

Note that we look at the many-to-one relationship between children and fathers as a single-valued fact about a child but a multi-valued fact about a father.

In a sense, fourth and fifth normal forms are also about composite keys. These normal forms attempt to **minimize the number of fields involved in a composite key**, as suggested by the examples to follow.

Fourth Normal Form

Whereas the second, third, and Boyce-Codd normal forms are concerned with functional dependencies, 4NF is concerned with a more general type of dependency known as a multivalued dependency.

- Under fourth normal form, a tuple should not contain **two or more independent** multivalued facts about an entity. In addition, the record must satisfy third normal form.

- Fourth normal form (4NF) is introduced by Ronald Fagin in 1977, 4NF is the next level of normalization after Boyce-Codd normal form (BCNF).

Consider following relation to understand the definition.

programmer (emp_name, qualifications, languages)

This relation has no non-key attributes because its only key is {emp_name, qualifications, languages}. Therefore it meets all normal forms up to BCNF.

But it contains **two many-to-many** relationships.
emp_name →→ qualification emp_name →→ languages

Here qualifications is independent of languages. So this relation features **two independent** non-trivial multivalued dependencies on the {emp_name} attribute (which is not a superkey).

Under fourth normal form, these two relationships should not be represented in a single relation. They should be decomposed into two relations (empid,skill) and (empid,language).

Consider following three schemes of storing the records of this relation. The features of these schemes are shown in right side.

emp_name	qualifications	languages
SMITH	B.Sc	FORTRAN
SMITH	B.Sc	COBOL
SMITH	B.Sc	PASCAL
SMITH	Dip.CS	FORTRAN
SMITH	Dip.CS	COBOL
SMITH	Dip.CS	PASCAL

A "cross-product" form, where for each employee, there must be a record for every possible pairing of one of his qualifications with one of his languages.

emp_name	qualifications	languages
SMITH	B.Sc	NULL
SMITH	Dip.CS	NULL
SMITH	NULL	FORTRAN
SMITH	NULL	COBOL
SMITH	NULL	PASCAL

The record contains either qualification or language but not both. It leads to ambiguities regarding the meanings of blank fields. A blank qualifications could mean the person has no qualifications, or the field is not applicable to this employee, or the data is unknown, or, as in this case, the data may be found in another record.

emp_name	qualifications	languages
SMITH	B.Sc	FORTRAN
SMITH	Dip.CS	COBOL
SMITH	NULL	PASCAL

Minimal no. of record with NULL values

All the above schemes are violating fourth normal form and following anomalies may arise:

- If there are repetitions, then updates have to be done in multiple records, and they could become inconsistent.
- Insertion of a new qualification may involve looking for a record with a blank language, or inserting a new record with a possibly blank language, or inserting multiple records pairing the new qualification with some or all of the languages.
- Deletion of a qualification may involve blanking out the qualification field in one or more records , or deleting one or more records, coupled with a check that the last mention of some language hasn't also been deleted.

Fourth normal form minimizes such anomalies. The two many-to-many relationships, emp_name→→qualifications and emp_name→→languages, are "independent" (there is no direct connection between qualifications and languages. So decompose it into two relations to make it 4NF.

emp_name	qualifications
SMITH	B.Sc
SMITH	Dip.CS

emp_name	languages
SMITH	FORTRAN
SMITH	COBOL
SMITH	PASCAL

Other Example :

Pizza Delivery Permutations		
Restaurant	**Pizza Variety**	**Delivery Area**
A1 Pizza	Thick Crust	Springfield
A1 Pizza	Thick Crust	Shelbyville
A1 Pizza	Thick Crust	Capital City
A1 Pizza	Stuffed Crust	Springfield
A1 Pizza	Stuffed Crust	Shelbyville
A1 Pizza	Stuffed Crust	Capital City
Elite Pizza	Thin Crust	Capital City
Elite Pizza	Stuffed Crust	Capital City
Vincenzo's Pizza	Thick Crust	Springfield
Vincenzo's Pizza	Thick Crust	Shelbyville
Vincenzo's Pizza	Thin Crust	Springfield
Vincenzo's Pizza	Thin Crust	Shelbyville

- Each row indicates that a given restaurant can deliver a given variety of pizza to a given area.
- The table has no non-key attributes because its only key is {Restaurant, Pizza Variety, Delivery Area}. Therefore it meets all normal forms up to BCNF.
- If we assume, however, that pizza varieties offered by a restaurant are not affected by delivery area (independent of each other), then it does not meet 4NF.
- The dependencies are:
  {Restaurant} →→ {Pizza Variety}
  {Restaurant} →→ {Delivery Area}

To eliminate the possibility of anomalies, we must place the facts about varieties offered into a different table from the facts about delivery areas, yielding two tables that are both in 4NF:

Varieties By Restaurant	
Restaurant	**Pizza Variety**
A1 Pizza	Thick Crust
A1 Pizza	Stuffed Crust
Elite Pizza	Thin Crust
Elite Pizza	Stuffed Crust
Vincenzo's Pizza	Thick Crust
Vincenzo's Pizza	Thin Crust

Delivery Areas By Restaurant	
Restaurant	**Delivery Area**
A1 Pizza	Springfield
A1 Pizza	Shelbyville
A1 Pizza	Capital City
Elite Pizza	Capital City
Vincenzo's Pizza	Springfield
Vincenzo's Pizza	Shelbyville

4NF in practice

A 1992 paper by Margaret S. Wu notes that the teaching of database normalization typically stops short of 4NF, perhaps because of a belief that tables violating 4NF (but meeting all lower normal forms) are rarely encountered in business applications. This belief may not be accurate, however. But a study of forty organizational databases shows that over 20% contained one or more tables that violated 4NF while meeting all lower normal forms.

Join Dependency and Fifth Normal form

Suppose we had a relation R(A,B,C) and we decomposed it into two relations S1(A,B) and S2(B,C). Then we'd like the decomposition to have the property that the natural join of the projected relations **$\Pi_{A,B}$(R) and $\Pi_{B,C}$(R)** would produce R. Essentially, we'd like to be able to retrieve the original R from the two relations in decomposition S1,S2

This won't always happen. Suppose we had this instance of R as shown below and we projected it to S_1 and S_2 as shown below:

Original R

A	B	C
1	2	3
2	2	4

$S_1(A,B)$

A	B
1	2
2	2

$S_2(B,C)$

B	C
2	3
2	4

the join $S_1 \bowtie S_2$ we'd have

A	B	C
1	2	3
1	2	4
2	2	3
2	2	4

Notice that the join contains the original tuples (1,2,3), 2,2,4) (1,2,3),(2,2,4) of R.

Two **spurious tuples** (1,2,4),(2,2,3) are also introduced which were not in the original relation R.

This is a situation we certainly don't want to have, which is why any such decomposition should satisfy the join dependency.

A relation R satisfies **join dependency** (R1, R2, ..., Rn) if and only if R is equal to the join of R1, R2, ..., Rn where R_i are subsets of the set of attributes of R.

Join Dependency exists where spurious rows are generated when tables are reunited through a natural join operation.

A relation decomposed into two relations must have **loss-less join Property**, which ensures that no spurious or extra tuples are generated, when relations are reunited through a natural join.

Fifth normal form

- The normal forms discussed so far required that the given relation R if not in the given normal form be decomposed in two relations to meet the requirements of the normal form.
- After reaching 4NF in this process, a relation can have problems like redundant information and update anomalies but cannot be decomposed in two relations to remove the problems.
- In such cases it may be possible to decompose the relation in three or more relations using the 5NF. Fifth normal form (5NF), also known as **Project-join normal form (PJ/NF).**
- The fifth normal form deals with join-dependencies which is a generalization of the MVD. The aim of fifth normal form is to have relations that cannot be decomposed further

A relation R is in 5NF if and only if it satisfies following conditions:

- R should be already in 4NF.
- It cannot be further non loss decomposed (join dependency) A relation in 5NF cannot be constructed from several smaller relations.

Example :

Consider the relation DSubStud, which lists the subjects & students in a department

Department	Subject	Student
Comp Sc.	CP1000	John Smith
Mathematics	MA1000	John Smith
Comp Sc.	CP2000	Arun Kumar
Comp Sc.	CP3000	Ron Roberts
Physics	PH1000	Raymond Craw
Chemistry	CH2000	Albert Garcia

It has multivalued dependencies Department $\rightarrow\rightarrow$Subject Department $\rightarrow\rightarrow$ Student

To make it 4NF split into 2 tables **DSub** & **DStud**

DStud

Department	Student
Comp Sc	John Smith
Comp Sc	Arun Kumar
Comp Sc	Ron Roberts
Mathematics	John Smith
Physics	Raymond Craw
Chemistry	Albert Garcia

DSub

Department	Subject
Comp Sc	CP1000
Comp Sc	CP2000
Comp Sc	CP3000
Mathematics	MA1000
Physics	PH1000
Chemistry	CH2000

However when we try to join these tables DSub & DStud

select * from dsub a, dstud b where a.dept =b.dept;

a.Department	a.Subject	b.Department	b.Student
Chemistry	CH2000	Chemistry	Albert Garcia
Comp Sc..	CP1000	Comp Sc	John Smith
Comp Sc..	CP2000	Comp Sc	John Smith
Comp Sc.	CP3000	Comp Sc.	John Smith
Comp Sc..	CP1000	Comp Sc	Arun Kumar
Comp Sc.	CP2000	Comp Sc.	Arun Kumar
Comp Sc..	CP3000	Comp Sc	Arun Kumar
Comp Sc.	CP1000	Comp Sc.	Ron Roberts
Comp Sc..	CP2000	Comp Sc	Ron Roberts
Comp Sc.	CP3000	Comp Sc.	Ron Roberts
Mathematics	MA1000	Mathematics	John Smith
Physics	PH1000	Physics	Raymond Craw

So Join gives 12 rows rather than the original 6

- Subject & Student are not independent
- So require to take to **Fifth Normal Form**

So Decompose into 3 tables

DSub & DStud as before plus **Substud**

Substud

Subject	Student
CP1000	John Smith
MA1000	John Smith
CP2000	Arun Kumar
CP3000	Ron Roberts
PH1000	Raymond Craw
CH2000	Albert Garcia

So join the 3 tables

select a.dept, c.subject, b.student
from dsub a, dstud b, substud c
where a.dept =b.dept and b.student = c.student
and a.subject = c.subject;

This table helps to eliminate the unwanted rows

DEPARTMENT	SUBJECT	STUDENT
Chemistry	CH2000	Albert Garcia
Comp Sc.	CP1000	John Smith
Comp Sc.	CP2000	Arun Kumar
Comp Sc.	CP3000	Ron Roberts
Mathematics	MA1000	John Smith
Physics	PH1000	Raymond Craw

Here we have decomposed into three relations **DSub**, **DStud** and **Substud** to make it 5NF. As shown above if we join theme no unwanted rows will be generated. So convert to 5NF by introducing the table SubStud to cater for the dependency between Subject & Student.

Roughly speaking, we may say that a record type is in fifth normal form when its information content cannot be reconstructed from several smaller record types, i.e., from record types each having fewer fields than the original record. The case where all the smaller records have the same key is excluded. If a record type can only be decomposed into smaller records which all have the same key, then the record type is considered to be in fifth normal form without decomposition. A record type in fifth normal form is also in fourth, third, second, and first normal forms.

Only in rare situations does a 4NF table not conform to 5NF.

Sixth normal form (Not in Syllabus)

Some authors use the term sixth normal form differently, namely, as a synonym for Domain/key normal form (DKNF).

Domain/key normal form (DKNF) or 6NF is a normal form used in database normalization which requires that the database contains no constraints other than domain constraints and key constraints.

- A domain constraint specifies the permissible values for a given attribute,
- A key constraint specifies the attributes that uniquely identify a row in a given table.
- The domain/key normal form is achieved when every constraint on the relation is a logical consequence of the definition of keys and domains, and enforcing key and domain restraints and conditions causes all constraints to be met.

The reason to use domain/key normal form is to avoid having general constraints in the database that are not clear domain or key constraints. General constraints would normally require special database programming in the form of stored procedures that are expensive to maintain and expensive for the database to execute. Therefore general constraints are split into domain and key constraints.

Successfully building a domain/key normal form database remains a difficult task, even for experienced database programmers. Thus, while the domain/key normal form eliminates the problems found in most databases, it tends to be the **most costly normal form** to achieve.

Example

A violation of DKNF occurs in the following table:

Wealthy Person		
Wealthy Person	**Wealthy Person Type**	**Net Worth in Dollars**
Steve	Eccentric Millionaire	124,543,621
Roderick	Evil Billionaire	6,553,228,893
Katrina	Eccentric Billionaire	8,829,462,998
Gary	Evil Millionaire	495,565,211

- Assume that the domain for Wealthy Person consists of the names of all wealthy people in a pre-defined sample of wealthy people;
- the domain for Wealthy Person Type consists of the values 'Eccentric Millionaire', 'Eccentric Billionaire', 'Evil Millionaire', and 'Evil Billionaire';
- the domain for Net Worth in Dollars consists of all integers greater than or equal to 1,000,000.)

There is a constraint linking Wealthy Person Type to Net Worth in Dollars, even though we cannot deduce one from the other. The constraint dictates followings :

- Eccentric Millionaire or Evil Millionaire will have a net worth of 1,000,000 to 999,999,999 inclusive,
- while an Eccentric Billionaire or Evil Billionaire will have a net worth of 1,000,000,000 or higher.

This constraint is neither a domain constraint nor a key constraint; therefore we cannot rely on domain constraints and key constraints to guarantee that an inconsistent Wealthy Person Type / Net Worth in Dollars combination does not make its way into the database.

The DKNF violation could be eliminated by altering the Wealthy Person Type domain to make it consist of just two values, 'Evil' and 'Eccentric' (the wealthy person's status as a millionaire or billionaire is implicit in their Net Worth in Dollars, so no useful information is lost).

Wealthy Person		
Wealthy Person	**Wealthy Person Type**	**Net Worth in Dollars**
Steve	Eccentric	124,543,621
Roderick	Evil	6,553,228,893
Katrina	Eccentric	8,829,462,998
Gary	Evil	495,565,211

Wealthiness Status		
Status	**Minimum**	**Maximum**
Millionaire	1,000,000	999,999,999
Billionaire	1,000,000,000	999,999,999,999

After achieving the programs won't need to make any logical decisions, they'll all be defined by the data.

"DKNF" is a theoretical ideal, but for most practical purposes it is ridiculous. By putting all the logic into the data definitions you certainly simplify the programming; however, you wind up with too many attributes in all the tuples (records) for the sake of saving a few processes on only a few of them. It's the old "space vs. time" debate - Waste file space in order to simplify and speed up programs.

Usage

The sixth normal form is currently being used in some data warehouses where the benefits outweigh the drawbacks.