Functional Dependency and Normalization for Relational Databases

Introduction:

Relational database design ultimately produces a set of relations. The implicit goals of the design activity are: *information preservation and minimum redundancy*.

Informal Design Guidelines for Relation Schemas

Four *informal guidelines* that may be used as *measures to determine the quality* of relation schema design:

- Making sure that the semantics of the attributes is clear in the schema
- Reducing the redundant information in tuples
- Reducing the NULL values in tuples
- Disallowing the possibility of generating spurious tuples

Imparting Clear Semantics to Attributes in Relations

The **semantics** of a relation refers to its meaning resulting from the interpretation of attribute values in a tuple. The relational schema design should have a clear meaning.

Guideline 1

1. Design a relation schema so that it is easy to explain.

2. Do not combine attributes from multiple entity types and relationship types into a single relation.

Redundant Information in Tuples and Update Anomalies

One goal of schema design is to minimize the storage space used by the base relations (and hence the corresponding files).

Grouping attributes into relation schemas has a significant effect on storage space

Storing natural joins of base relations leads to an additional problem referred to as **update anomalies**. These are: insertion anomalies, deletion anomalies, and modification anomalies.

Insertion Anomalies happen:

- when insertion of a new tuple is not done properly and will therefore can make the database become inconsistent.
- When the insertion of a new tuple introduces a NULL value (for example a department in which no employee works as of yet). This will violate the integrity constraint of the table since ESsn is a primary key for the table.

Deletion Anomalies:

The problem of deletion anomalies is related to the second insertion anomaly situation just discussed.

Example: If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost from the database.

Modification Anomalies happen if we fail to update all tuples as a result in the change in a single one.

Example: if the manager changes for a department, all employees who work for that department must be updated in all the tables.

It is easy to see that these three anomalies are undesirable and cause difficulties to maintain consistency of data as well as require unnecessary updates that can be avoided; hence

Guideline 2

Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations.

If any anomalies are present, note them clearly and make sure that the programs that update the database will operate correctly.

The second guideline is consistent with and, in a way, a restatement of the first guideline.

NULL Values in Tuples

Fat Relations: A relation in which too many attributes are grouped. If many of the attributes do not apply to all tuples in the relation, we end up with many NULLs in those tuples. This can waste space at the storage level and may also lead to problems with understanding the meaning of the attributes and with specifying JOIN operations at the logical level.

Another problem with NULLs is how to account for them when aggregate operations such as COUNT or SUM are applied.

SELECT and JOIN operations involve comparisons; if NULL values are present, the results may become unpredictable. Moreover, NULLs can have multiple interpretations, such as the following:

- The attribute *does not apply* to this tuple. For example, Visa_status may not apply to U.S. students.
- The attribute value for this tuple is *unknown*. For example, the Date_of_birth may be unknown for an employee.
- The value is *known but absent*; that is, it has not been recorded yet. For example, the Home_Phone_Number for an employee may exist, but may not be available and recorded yet.

Having the same representation for all NULLs compromises the different meanings they may have. Therefore, we may state another guideline.

Guideline 3

As much as possible, avoid placing attributes in a base relation whose values may frequently be NULL.

If NULLs are unavoidable, make sure that they apply in exceptional cases only.

For example, if only 15 percent of employees have individual offices, there is little justification for including an attribute Office_number in the EMPLOYEE relation; rather, a relation EMP_OFFICES(Essn, Office_number) can be created

Generation of Spurious Tuples

Often, we may elect to split a “fat” relation into two relations, with the intention of joining them together if needed. However, applying a NATURAL JOIN may not yield the desired effect. On the contrary, it will generate many more tuples and we cannot recover the original table.

Guideline 4

Design relation schemas so that they can be joined with equality conditions on attributes that are appropriately related (primary key, foreign key) pairs in a way that guarantees that no spurious tuples are generated.

Avoid relations that contain matching attributes that are not (foreign key, primary key) combinations because joining on such attributes may produce spurious tuples.

Summary and Discussion of Design Guidelines

We proposed informal guidelines for a good relational design. The problems we pointed out, which can be detected without additional tools of analysis, are as follows:

- Anomalies that cause redundant work to be done during insertion into and modification of a relation, and that may cause accidental loss of information during a deletion from a relation
- Waste of storage space due to NULLs and the difficulty of performing selections, aggregation operations, and joins due to NULL values
- Generation of invalid and spurious data during joins on base relations with matched attributes that may not represent a proper (foreign key, primary key) relationship

The strategy for achieving a good design is to decompose a badly designed relation appropriately.

Functional Dependencies

The single most important concept in relational schema design theory is that of a functional dependency.

Definition of Functional Dependency

A functional dependency is a constraint between two sets of attributes from the database. Suppose that our relational database schema has n attributes $A1, A2, ..., An$.

If we think of the whole database as being described by a single **universal** relation schema $R = \{A1, A2, ... , An\}$.

A **functional dependency**, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R, *such that* any two tuples $t1$ and $t2$ in r that have $t1[X] = t2[X]$, they must also have $t1[Y] = t2[Y]$.

This means that the values of the Y component of a tuple in r depend on, or are *determined by,* the values of the X component; We say that the values of the X component of a tuple uniquely (or **functionally**) *determine* the values of the Y component.

We say that there is a functional dependency from X to Y, or that Y is **functionally dependent** on X.

Functional dependency is represented as **FD** or **f.d.** The set of attributes *X* is called the **left-hand side** of the FD, and *Y* is called the **right-hand side**.

X functionally determines *Y* in a relation schema *R* if, and only if, whenever two tuples of *r*(*R*) agree on their *X*-value, they must necessarily agree on their *Y*-value.
If a constraint on *R* states that there cannot be more than one tuple with a given *X*-value in any relation instance *r*(*R*)—that is, *X* is a **candidate key** of *R*— this implies that $X \rightarrow Y$ for any subset of attributes *Y* of *R.*
If *X* is a candidate key of *R*, then $X \rightarrow R$.

If $X \rightarrow Y$ in *R*, this does not imply that $Y \rightarrow X$ in *R*.

A functional dependency is a property of the **semantics** or **meaning of the attributes**.
Whenever the semantics of two sets of attributes in *R* indicate that a functional dependency should hold, we specify the dependency as a constraint.

Legal Relation States:

Relation extensions *r*(*R*) that satisfy the functional dependency constraints are called **legal relation states** (or **legal extensions**) of *R*.

Functional dependencies are used to describe further a relation schema *R* by specifying constraints on its attributes that must hold *at all times.*

Certain FDs can be specified without referring to a specific relation, but as a property of those attributes given their commonly understood meaning.

For example, {State, Driver_license_number} → Ssn should hold for any adult in the United States and hence should hold whenever these attributes appear in a relation.

Consider the relation schema EMP_PROJ from the semantics of the attributes and the relation, we know that the following functional dependencies should hold:

a. Ssn→Ename

b. Pnumber →{Pname, Plocation}

c. {Ssn, Pnumber}→Hours

A functional dependency is a *property of the relation schema R*, not of a particular legal relation state *r* of *R*. Therefore, an FD *cannot* be inferred automatically from a given relation extension *r* but must be defined explicitly by someone who knows the semantics of the attributes of *R*.

Teacher	Course	Text
Smith	Data Structures	Bartram
Smith	Data Management	Martin
Hall	Compilers	Hoffman
Brown	Data Structures	Horowitz

Example:

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

The following FDs *may hold* because the four tuples in the current extension have no violation of these constraints:

$B \rightarrow C$; $C \rightarrow B$; $\{A, B\} \rightarrow C$; $\{A, B\} \rightarrow D$; and $\{C, D\} \rightarrow B$

However, the following *do not* hold because we already have violations of them in the given extension:

$A \rightarrow B$ (tuples 1 and 2 violate this constraint);

$B \rightarrow A$ (tuples 2 and 3 violate this constraint);

$D \rightarrow C$ (tuples 3 and 4 violate it).

Diagrammatic notation for displaying FDs:

Each FD is displayed as a horizontal line. The left-hand-side attributes of the FD are connected by vertical lines to the line representing the FD, while the right-hand-side attributes are connected by the lines with arrows pointing toward the attributes.

Normal Forms Based on Primary Keys

Normalization of data is a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of

(1) minimizing redundancy and

(2) minimizing the insertion, deletion, and update anomalies.

It can be considered as a “filtering” or “purification” process to make the design have successively better quality.

We assume that a set of functional dependencies is given for each relation, and that each relation has a designated primary key.

Each relation is then evaluated for adequacy and decomposed further as needed to achieve higher normal forms, using the normalization theory.

We focus on the first three normal forms for relation schemas and the intuition behind them, and discuss how they were developed historically.

More general definitions of these normal forms, which take into account all candidate keys of a relation rather than just the primary key.

Normalization of Relations

The normalization process, as first proposed by Codd (1972a), takes a relation schema through a series of tests to *certify* whether it satisfies a certain **normal form**.

The process, which proceeds in a top-down fashion by evaluating each relation against the criteria for normal forms and decomposing relations as necessary, can thus be considered as *relational design by analysis.*

Initially, Codd proposed three normal forms, which he called first, second, and third normal form.

A stronger definition of 3NF—called Boyce-Codd normal form (BCNF)—was proposed later by Boyce and Codd. All these normal forms are based on a single analytical tool: the functional dependencies among the attributes of a relation.

The normalization procedure provides database designers with:
A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes.
A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be **normalized** to any desired degree

Definition.
The **normal form** of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized.
Normal forms, when considered *in isolation* from other factors, do not guarantee a good database design. It is generally not sufficient to check separately that each relation schema in the database is in a given normal form.
Rather, the process of normalization through decomposition must also confirm the existence of additional properties that the

relational schemas, taken together, should possess. These would include two properties:

The **nonadditive join or lossless join property**, which guarantees that the spurious tuple generation problem does not occur with respect to the relation schemas created after decomposition.

The **dependency preservation property**, which ensures that each functional dependency is represented in some individual relation resulting after decomposition.

The nonadditive join property is extremely critical and **must be achieved at any cost**.

Practical Use of Normal Forms

Most practical design projects acquire existing designs of databases from previous designs, designs in legacy models, or from existing files.

Normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties stated previously.

Although several higher normal forms have been defined, database design as practiced in industry today pays particular attention to normalization only up to 3NF, BCNF, or at most 4NF.

Another point worth noting is that the database designers *need not* normalize to the highest possible normal form. Relations may be left in a lower normalization status, such as 2NF, for performance reason.

Denormalization is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.

Definitions of Keys and Attributes

Participating in Keys

Definition: A **superkey** of a relation schema $R = \{A1, A2, ..., An\}$ is a set of attributes $S \subseteq R$ with the property that no two tuples $t1$ and $t2$ in any legal relation state r of R will have $t1[S] = t2[S]$.

A **key** K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey anymore.

The difference between a key and a superkey is that a key has to be *minimal;* that is, if we have a key

$K = \{A1, A2, ..., Ak\}$ of R, then $K - \{Ai\}$ is not a key of R for any Ai, $1 \leq i \leq k$

{Ssn} is a key for EMPLOYEE, whereas {Ssn}, {Ssn, Ename},

{Ssn, Ename, Bdate}, and any set of attributes that includes Ssn are all superkeys.

If a relation schema has more than one key, each is called a **candidate key**.

One of the candidate keys is *arbitrarily* designated to be the **primary key**, and the others are called secondary keys.

In a practical relational database, each relation schema must have a primary key. If no candidate key is known for a relation, the entire relation can be treated as a default superkey. In the Table EMPLOYEE, {Ssn} is the only candidate key for EMPLOYEE, so it is also the primary key.

Definition:

An attribute of relation schema *R* is called a **prime attribute** of *R* if it is a member of *some candidate key* of *R*.

An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key, both Ssn and Pnumber are prime attributes of WORKS_ON, whereas other attributes of WORKS_ON are nonprime.

We now present the first three normal forms: 1NF, 2NF, and 3NF. As we shall see, 2NF and 3NF attack different problems.

First Normal Form

First normal form (1NF) is now considered to be part of the formal definition of a relation in the basic (flat) relational model.
It states that:

1. the domain of an attribute must include only *atomic* (simple, indivisible) *values* and
2. that the value of any attribute in a tuple must be a *single value* from the domain of that attribute.

Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a *single tuple.* In other words, 1NF disallows *relations within relations* or *relations as attribute values within tuples*.
The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values**.

Consider the DEPARTMENT relation schema, whose primary key is Dnumber, and suppose that we extend it by including the Dlocations attribute.
We assume that each department can have *a number of* locations.
As we can see, this is not in 1NF because Dlocations is not an atomic attribute. There are two ways we look at the Dlocations attribute:

The domain of Dlocations contains atomic values, but some tuples can have a set of these values. In this case, Dlocations is not functionally dependent on the primary key Dnum.

First normal form also disallows multi-valued attributes that are themselves composite. These are called **nested relations** because each tuple can have a relation *within it.*
This procedure can be applied recursively to a relation with multiple-level nesting to **unnest** the relation into a set of 1NF relations. This is useful in converting an unnormalized relation schema with many levels of nesting into 1NF relations.

Second Normal Form
Second normal form (2NF) is based on the concept of *full functional dependency.*
Functional Dependency:
The attribute B is fully functionally dependent on the attribute A if each value of A determines one and only one value of B.
Example: PROJ_NUM→ PROJ_NAME
In this case, the attribute PROJ_NUM is known as the determinant attribute and the attribute PROJ_NAME is known as the dependent attribute.

Generalized Definition:

Attribute A determines attribute B (that is B is functionally dependent on A) if all of the rows in the table that agree in value for attribute A also agree in value for attribute B.

Fully functional dependency (composite key)

If attribute B is functionally dependent on a composite key A but not on any subset of that composite key, the attribute B is fully functionally dependent on A.

Partial Dependency:

When there is a functional dependence in which the determinant is only part of the primary key, then there is a partial dependency.

For example if (A, B) → (C, D) and B→ C and (A, B) is the primary key, then the functional dependence B→ C is a partial dependency.

{Ssn, Pnumber} → Hours is a full dependency

(neither Ssn → Hours nor Pnumber→Hours holds).

However, the dependency {Ssn, Pnumber}→Ename is partial because Ssn→Ename holds.

Transitive Dependency:

When there are the following functional dependencies such that X→Y, Y→ Z and X is the primary key, then X→ Z is a transitive dependency because X determines the value of Z via Y. Whenever a functional dependency is detected amongst non-prime, there is a transitive dependency.

Definition. A relation schema *R* is in 2NF if every nonprime attribute *A* in *R* is *fully functionally dependent* on the primary key of *R*.

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key.

If the primary key contains a single attribute, the test need not be applied at all.

If a relation schema is not in 2NF, it can be *second normalized* or *2NF normalized* into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent.

Third Normal Form

Third normal form (3NF) is based on the concept of *transitive dependency*.

A functional dependency $X \rightarrow Y$ in a relation schema *R* is a **transitive dependency** if there exists a set of attributes *Z* in *R*

that is neither a candidate key nor a subset of any key of *R*, and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold.

Definition. According to Codd's original definition, a relation schema *R* is in **3NF** if it satisfies 2NF *and* no nonprime attribute of *R* is transitively dependent on the primary key.

General Definitions of Second and Third Normal Forms

In general, we want to design our relation schemas so that they have neither partial nor transitive dependencies because these types of dependencies cause the update anomalies seen previously.

The steps for normalization into 3NF relations that we have discussed so far disallow partial and transitive dependencies on the *primary key.* The normalization procedure described so far is useful for analysis in practical situations for a given database where primary keys have already been defined.

Summary of Normal Forms Based on Primary Keys and Corresponding Normalization

Normal Form	Test	Remedy (Normalization)
First (1NF)	Relation should have no multivalued attributes or nested relations	Form new relations for each multivalued attribute or nested relation.
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.
Third (3NF)	Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes). That is, there should be no transitive dependency of a nonkey attribute on the primary key	Decompose and setup a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

As a general definition of **prime attribute**, an attribute that is part of *any candidate key* will be considered as prime. Partial and full functional dependencies and transitive dependencies will now be considered *with respect to all candidate keys* of a relation.
Prime attributes are part of any candidate key
Non-prime attribute are not.

General Definition of Second Normal Form

A relation schema *R* is in **second normal form (2NF)** if every nonprime attribute *A* in *R* is not partially dependent on *any* key of *R*.

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are *part of* the primary key. If the primary key contains a single attribute, the test need not be applied at all.

General Definition of Third Normal Form

A relation schema *R* is in **third normal form (3NF)** if, whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in *R*, either

(a) *X* is a superkey of *R*, or

(b) *A* is a prime attribute of *R*.

Interpreting the General Definition of Third Normal Form

A relation schema *R* violates the general definition of 3NF if a functional dependency $X \rightarrow A$ holds in *R* that does not meet either condition—meaning that it violates *both* conditions (a) and (b) of 3NF. This can occur due to two types of problematic functional dependencies:

A nonprime attribute determines another nonprime attribute. Here we typically have a transitive dependency that violates 3NF.

A proper subset of a key of *R* functionally determines a nonprime attribute. Here we have a partial dependency that violates 3NF (and also 2NF).

Therefore, we can state a **general alternative definition of 3NF** as follows:

Alternative Definition. A relation schema *R* is in 3NF if every nonprime attribute of *R* meets both of the following conditions:

- It is fully functionally dependent on every key of *R*.
- It is nontransitively dependent on every key of *R*.

Boyce-Codd Normal Form

Boyce-Codd normal form (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF.

Definition: A relation schema *R* is in **BCNF** if whenever a nontrivial functional dependency X→ A holds in R, then X is a superkey of R. In practice, most relation schemas that are in 3NF are also in BCNF.

Only if $X \rightarrow A$ holds in a relation schema *R* with *X* not being a superkey *and A* being a prime attribute will *R* be in 3NF but not in BCNF. Ideally, relational database design should strive to achieve BCNF or 3NF for every relation schema.

Conversion to First Normal Form:

A relational table must not contain repeating groups. A repeating group derives its name from the fact that a group of multiple entries of the same type can exist for any single key attribute occurrence.

If repeating groups do exist, they must be eliminated by making sure that each row defines a single entity.

Normalization starts with a simple three-step procedure:

Step 1: Eliminate the Repeating Groups:

1. Represent the data in a tabular format, where each cell has a single value and there are no repeating groups.
2. To eliminate repeating groups: eliminate the nulls by making sure that each repeating group contains appropriate data value.

Step 2: Identify the Primary Key:

To have a proper Primary Key, it should uniquely identify any attribute value.

In our example, we can see that PROJ_NUM value 15, identifies any one of 5 employees.

EMP_NUM can also identify multiple rows, since one employee can work in more than one project.

In this case, the only primary key possible is a combination of PROJ_NUM and EMP_NUM.

Step 3:

Identify all dependencies:

PROJ_NUM and EMP_NUM→ PROJ_NAME, EMP_NAME, JOB_CLASS, CHG_HOUR, HOURS.

Additional dependencies:

PROJ_NUM→ PROJ_NAME

EMP_NUM→ EMP_NAME, JOB_CLASS, CHG_HOUR

JOB_CLASS→ CHG_HOUR

This dependency exists between two nonprime attributes, which signals a transitive dependency.

Conversion to Second Normal Form:

Conversion to 2NF only occurs when the 1NF has a composite primary key. If the 1NF has a single-attribute primary key, then the table is automatically 2NF.

Step 1: Make new tables to Eliminate Partial Dependencies

For each component of the primary key that acts as a determinant in a partial dependency, create a new table with a copy of that component as the primary key. It is also important that the determinant attribute remains in the original table because they

will be the foreign keys that will relate the new tables to the original one.

Step 2: Reassign Corresponding Dependent Attributes

Determine all attributes that are dependent in the partial dependencies. These are removed from the original table and placed in the new table with their determinant.

Any attributes that are dependent in a partial dependency will remain in the original table.

Now, we have 3 tables:

PROJECT(**PROJ_NUM,** PROJ_NAME)

EMPLOYEE(**EMP_NUM**, EMP_NAME, JOB_CLASS, CHG_HOURS)

ASSIGNMENT(**PROJ_NUM, EMP_NUM,** ASSIGN_HOURS)

Conversion to third Normal Form:

Step 1: Make new tables to eliminate transitive dependencies.

For every transitive dependency, write a copy if its determinant as a primary key for a new table. It is also important that the determinant remains in the original table to serve as a foreign key.

Step 2:

Identify the attributes that are dependent on each determinant and place them in the new tables with their determinant and remove them from their original table.

In our example, remove CHG_HOUR from EMPLOYEE

EMP_NUM→EMP_NAME, JOB_CLASS

So now our design becomes:

PROJECT(**PROJ_NUM,** PROJ_NAME)

EMPLOYEE(**EMP_NUM**, EMP_NAME, JOB_ID)

JOB(**JOB_ID**, JOB_CLASS**,** CHG_HOUR)

ASSIGNMENT(**PROJ_NUM, EMP_NUM,** ASSIGN_HOURS)

Consider the table below, describing a badly designed database. Follow the steps defined above and seen in class to make the design 3NF compliant.

StdNo	StdCity	StdClass	OfferNo	OffTerm	OffYear	EnrGrade	CourseNo	CrsDescr
S1	Seattle	JUN	01	FALL	2013	3.5	C1	DB
			02	FALL	2013	3.3	C2	VB
S2	Bothell	JUN	03	SPRING	2014	3.1	C3	OO
			02	FALL	2013	3.4	C2	VB