

1 Introduction to Normalization

Normalization is the process of organizing relational database tables to reduce unnecessary duplication, prevent inconsistent data, and make updates more reliable. The central idea is that each fact should be stored in an appropriate place, rather than repeated throughout the database.

Consider a school database that stores students, courses, and instructors. Suppose every registration record contains the student's name, the course name, and the instructor's name. If 30 students take the same course, the instructor's name might be repeated 30 times. If the instructor changes their name or the course moves to a different instructor, the database may require dozens of updates. Missing even one update can leave the database inconsistent.

Normalization helps us identify which facts belong together and which facts should be stored separately. Student information belongs to a student, course information belongs to a course, and a student's registration belongs to the relationship between that student and that course.

Normalization is based on the logical relationships between data, not merely on how many columns a table contains or how many duplicate values happen to appear in its current contents. A table with no duplicate values in a small sample may still have a poor design if its structure permits anomalies as the database grows.

The normal forms are a sequence of increasingly restrictive rules. First normal form (1NF) concerns the structure of values in rows and columns. Second normal form (2NF) addresses attributes that depend on only part of a composite key. Third normal form (3NF) addresses certain dependencies between non-key attributes. Boyce-Codd normal form (BCNF) strengthens the requirements on determinants. Higher normal forms address additional kinds of redundancy.

2 Understanding Keys

Before studying normalization, we need to understand how a relational table identifies its rows.

2.1 What Is a Key?

A key is a set of attributes that can uniquely identify a row in a relation. A key must distinguish one row from every other row allowed by the database design.

Imagine a classroom in which several children have the same first name. The first name alone cannot identify a child. If two children are both named Alex, the value `first_name = 'Alex'` identifies both of them.

Suppose every student has a unique student number. Then `student_id` can identify each student, even when several students share the same name.

```
Students(student_id, first_name, last_name)
```

For example:

student_id	first_name	last_name
101	Alex	Smith
102	Alex	Jones
103	Sam	Smith

The student number uniquely identifies each row. The first name does not. The combination of first and last names does not necessarily identify a student either, because two students might share the same full name.

A key is therefore not simply a column that happens to contain different values in today's data. It represents a uniqueness rule that must hold for all valid states of the database.

2.2 Superkeys and Candidate Keys

A superkey is any set of attributes that uniquely identifies a row. A candidate key is a superkey that is minimal: no attribute can be removed from it while preserving uniqueness.

Consider a table:

```
Students(student_id, email, first_name, last_name)
```

Assume that every student has a unique student ID and a unique email address.

Both of the following are candidate keys:

- `student_id`
- `email`

Each one is sufficient to identify a student. Neither needs another attribute added to it.

The set (`student_id`, `email`) is also a superkey because it uniquely identifies students. However, it is not a candidate key: the student ID alone already identifies the row, and so does the email address alone. The combination contains unnecessary attributes.

This distinction is important. Candidate keys are minimal identifiers; superkeys may contain additional attributes that are not needed.

2.3 Composite Keys

Sometimes no single attribute identifies a row, but a combination does. Such a combination can be a candidate key.

Consider student registrations:

<code>Registrations(student_id, course_id, grade)</code>
--

A student may take many courses, and each course may have many students. Neither `student_id` nor `course_id` alone identifies a registration. However, if a student can register for each course only once, the pair

(`student_id`, `course_id`)

is a candidate key.

For example, student 101 might register for courses 10 and 20. These are two distinct registrations because the course IDs differ. Another student can also register for course 10 because the student IDs differ.

A key made from more than one attribute is called a composite key. Composite keys are especially important in normalization because some attributes may depend on only one part of the key.

2.4 Primary Keys and Candidate Keys

A table can have multiple candidate keys, but a database designer typically selects one of them as the primary key. The remaining candidate keys are alternate keys and can be enforced with constraints such as `UNIQUE`.

For example, a student table might use `student_id` as its primary key while also enforcing uniqueness on `email`. Choosing a primary key does not make the other candidate keys irrelevant to normalization. Normal forms depend on all candidate keys, not only the primary key chosen for implementation.

A practical warning: an artificial identifier, such as an auto-incrementing integer, does not automatically solve normalization problems. A table can have a perfectly unique primary key and still store redundant facts or contain undesirable dependencies between other columns.

3 Functional Dependencies

3.1 The Basic Idea

A functional dependency describes a rule about the relationship between attributes in a relation. Written as

$$X \rightarrow Y$$

it means that whenever two rows have the same values for all attributes in X , they must also have the same values for all attributes in Y .

We say that X functionally determines Y .

The simplest intuitive example is a student ID. If every student ID belongs to exactly one student, then knowing the student ID tells us the student's name:

$$\text{student_id} \rightarrow \text{student_name}$$

If student ID 101 belongs to Alex Smith, every valid row for student 101 must use the same student name, wherever that dependency is intended to hold.

A functional dependency is a rule about all valid data, not just an observation about the rows currently stored. If a sample contains only one student named Alex, that does not establish that first name determines last name. A new student named Alex could have a different last name.

3.2 An Example from Everyday Life

Imagine a toy store. Every product has a unique product ID, and each product ID identifies one product name and one standard price.

$$\text{product_id} \rightarrow (\text{product_name}, \text{standard_price})$$

If product ID 500 means “Toy Car” with a standard price of \$12.00, every row describing product 500 must agree on those facts.

However, product name does not necessarily determine product ID. A store could sell different products with the same name, or it could have two suppliers selling equivalent items under separate identifiers. Whether product name determines product ID depends on the actual business rules.

This distinction matters because a functional dependency is not a statement about what seems likely. It is a statement about what the database's rules guarantee.

3.3 Functional Dependencies Can Have Multiple Attributes

A determinant can consist of several attributes. For example, in a course registration system, the grade a student receives depends on both the student and the course:

$$(\text{student_id}, \text{course_id}) \rightarrow \text{grade}$$

The student ID alone does not determine the grade because the student may receive different grades in different courses. The course ID alone does not determine the grade because different students may receive different grades in the same course.

The combination determines the grade because it identifies the particular student-course registration.

Functional dependencies are directional. The fact that $X \rightarrow Y$ does not imply that $Y \rightarrow X$. A student ID determines a student name, but a student name usually does not determine a student ID.

3.4 Functional Dependencies and Business Rules

Functional dependencies come from the meaning of the data. Consider a table containing employee IDs, department IDs, and department names.

If every department ID refers to exactly one department name, then

$$\text{department_id} \rightarrow \text{department_name}$$

This rule remains true regardless of how many employees work in each department.

On the other hand, the dependency

$$\text{employee_id} \rightarrow \text{department_id}$$

is valid only if each employee belongs to exactly one department in the modeled relation. If employees can belong to multiple departments, the dependency may not hold. If the table records department assignments over time, the employee ID alone may not determine the department ID.

Always clarify what one row represents and what the business rules guarantee before writing functional dependencies.

3.5 Trivial and Nontrivial Dependencies

A functional dependency $X \rightarrow Y$ is trivial if every attribute in Y is already contained in X . For example,

$$(\text{student_id}, \text{course_id}) \rightarrow \text{student_id}$$

is trivial. Knowing a pair necessarily tells us the value of each component of the pair.

A dependency is nontrivial when Y contains an attribute that is not already in X . For example,

$$\text{student_id} \rightarrow \text{student_name}$$

is nontrivial if the student name is not part of the determinant.

Normalization focuses on the meaningful dependencies that reveal how facts are related and where those facts should be stored.

4 First Normal Form (1NF)

4.1 Definition

A relation is in first normal form when each attribute value is atomic in the context of the chosen relational model, and each row and column has a well-defined meaning. A column should not contain a repeating group of values that the design expects to manipulate as separate items.

For introductory database design, the practical rule is simple: each cell should contain one value for that attribute, rather than a list of independently meaningful values.

4.2 An Example That Violates 1NF

Consider the following table:

student_id	student_name	courses
101	Alex	Database, Statistics
102	Sam	History

The **courses** column stores a list of courses in one cell. Finding all students taking Statistics, counting registrations, or joining registrations to course information becomes unnecessarily difficult. The database must interpret part of a text value as a separate logical item.

A better design represents each student-course registration as a separate row:

student_id	course
101	Database
101	Statistics
102	History

Each cell now contains one course value. The combination of `student_id` and `course` might identify a row if a student can register for a course only once. A course ID would usually be preferable to a course name in a real database.

4.3 What Does and Does Not Violate 1NF?

A table does not violate 1NF merely because a value contains spaces, punctuation, or multiple words. For example, a full name such as “Alex Smith” can be one atomic value if the application treats it as one attribute. Whether to split names into first and last names is a separate design decision.

A phone number stored as one string can also be atomic. A column containing three phone numbers separated by commas is generally not atomic if the numbers need to be queried and maintained independently.

Similarly, an address can be modeled as one value for some applications, but an application that needs to query cities, postal codes, and street names separately may benefit from storing those components separately. Atomicity is considered in the context of the database’s intended operations and data model.

A table may satisfy 1NF and still be poorly normalized. For example, repeating a department name in every employee row does not by itself violate 1NF. That redundancy is addressed by higher normal forms.

5 Second Normal Form (2NF)

5.1 The Basic Idea

Second normal form requires a relation to be in 1NF and to have no non-prime attribute that depends on a proper subset of any candidate key. A prime attribute is an attribute that belongs to at least one candidate key. A non-prime attribute belongs to no candidate key.

For an introductory treatment, the most common situation is a table with a composite candidate key. We ask whether an ordinary, non-key attribute depends on the entire key or only part of it.

If the key is (A, B) , then a dependency

$$A \rightarrow C$$

shows that C depends on only part of the key. If C is a non-prime attribute, this violates 2NF.

The underlying problem is that the table represents a relationship between A and B , but it also stores a fact that belongs to A alone. That fact is likely to be repeated across many rows.

5.2 An Example: Student Registrations

Consider this relation:

```
Registration(  
    student_id,  
    course_id,  
    student_name,  
    course_name,  
    grade  
)
```

Assume that a student may take many courses and each course may have many students. Each student can register for a course only once. The candidate key is

(student_id, course_id)

Suppose the following functional dependencies hold:

student_id $\rightarrow$ student_name
course_id $\rightarrow$ course_name
(student_id, course_id) $\rightarrow$ grade

The grade depends on the particular student and course together. Neither the student nor the course alone determines the grade.

However, the student name depends only on the student ID, and the course name depends only on the course ID. These are partial dependencies on parts of the composite key.

5.3 Why This Design Causes Problems

Suppose Alex takes five courses. Alex's name is repeated in five registration rows. If the name changes, five rows must be updated. If one row is missed, the database may contain two different names for the same student.

There is also an insertion problem. If the database stores student information only in the registration table, it may be impossible to insert a new student until the student registers for a course. Likewise, it may be impossible to store a new course until at least one student registers for it.

Finally, deleting the last registration for a student may accidentally remove the only stored copy of that student's name. These are update, insertion, and deletion anomalies.

5.4 Converting the Relation into 2NF

Separate facts about students, facts about courses, and facts about registrations into different relations:

<pre>Students(student_id, student_name) Courses(course_id, course_name) Registrations(student_id, course_id, grade)</pre>

The candidate key of **Students** is `student_id`. The candidate key of **Courses** is `course_id`. The candidate key of **Registrations** is the pair (`student_id`, `course_id`).

Now the student name is stored once per student, and the course name is stored once per course. The grade remains in the registration relation because it depends on the entire student-course key.

All three relations are in 2NF, assuming the stated dependencies and candidate keys. In particular, the registration relation has no non-prime attributes that depend on only part of its candidate key.

The foreign keys from **Registrations** to **Students** and **Courses** connect the relations. Normalization does not discard the relationships between facts; it represents them explicitly.

5.5 When Is 2NF Relevant?

If every candidate key consists of a single attribute, a relation cannot have a partial dependency on a proper subset of that key. Therefore, a relation in 1NF with only single-attribute candidate keys automatically satisfies 2NF.

This is why 2NF discussions often begin with composite keys. However, formal definitions refer to all candidate keys, not only the primary key selected by the database designer. A relation with multiple candidate keys must be checked against each of them.

6 Third Normal Form (3NF)

6.1 The Basic Idea

Third normal form requires a relation to be in 2NF and to satisfy an additional condition on functional dependencies. For every nontrivial functional dependency

$$X \rightarrow A$$

at least one of the following must hold:

- X is a superkey, or
- A is a prime attribute, meaning it belongs to some candidate key.

An intuitive way to understand the most common 3NF problem is to look for a non-key attribute that determines another non-key attribute. This often means that a fact about one entity is being stored repeatedly in rows about another entity.

6.2 An Example: Employees and Departments

Consider the relation:

```
Employees(  
    employee_id,  
    employee_name,  
    department_id,  
    department_name  
)
```

Assume each employee belongs to exactly one department, and each department has one name.

The functional dependencies are:

$$\begin{aligned} \text{employee_id} &\rightarrow (\text{employee_name}, \text{department_id}) \\ \text{department_id} &\rightarrow \text{department_name} \end{aligned}$$

The candidate key is **employee_id**. It determines the employee's name and department ID. Because the department ID determines the department name, we also have the implied dependency

$$\text{employee_id} \rightarrow \text{department_name}$$

This is a transitive dependency: the employee ID determines the department ID, which determines the department name.

The department name is not a property that independently belongs to every employee. It is a property of the department. Storing it in the employee table repeats the same department fact for every employee assigned to that department.

6.3 Why the Design Causes Problems

If 50 employees belong to the same department, the department name appears 50 times. Renaming the department requires updating all 50 rows. An incomplete update can cause inconsistent department names.

The table also makes it difficult to store a department before hiring its first employee. If the last employee in a department is deleted, the department's name may disappear from the database.

These problems occur even though the table is already in 2NF. The primary key has only one attribute, so there is no partial dependency on a subset of a composite key. The problem is instead a dependency between non-key attributes.

6.4 Converting the Relation into 3NF

Separate the department facts from the employee facts:

```
Employees(employee_id, employee_name, department_id)
```

```
Departments(department_id, department_name)
```

In **Employees**, the employee ID determines the employee's name and department ID. In **Departments**, the department ID determines the department name.

The `department_id` in **Employees** is a foreign key referencing **Departments**. It records which department an employee belongs to without duplicating the department's name for every employee.

Now a department's name is stored once in the department relation. Updating that name requires only one change, and a department can exist independently of its employees.

Both relations are in 3NF under the stated business rules. The decomposition preserves the functional dependencies and allows the original employee-and-department information to be recovered by joining the tables.

7 A Complete Worked Example: From 1NF to 2NF to 3NF

We will now examine a relation that requires both 2NF and 3NF decomposition. The example combines the ideas introduced earlier and demonstrates how functional dependencies guide the process.

7.1 The Original Relation

Suppose a company records which employees work on which projects. For each employee-project assignment, it stores the employee's name, the project name, the employee's department, and the department's office location.

The original relation is:

```
Assignment(  
    employee_id,  
    project_id,  
    employee_name,  
    project_name,  
    department_id,  
    department_name,  
    office_location,  
    hours_worked  
)
```

Assume the following business rules:

1. An employee can work on several projects.
2. A project can have several employees.
3. Each employee belongs to one department.
4. Each department has one name and one office location.
5. Each project has one project name.
6. The hours worked are recorded for each employee-project assignment.

Assume the relation contains one row per employee-project assignment. The candidate key is

(employee_id, project_id)

Neither attribute alone identifies an assignment. An employee can appear on several projects, and a project can have several employees.

7.2 Step 1: Identify the Functional Dependencies

From the business rules, we obtain:

$$\begin{aligned} \text{employee_id} &\rightarrow (\text{employee_name}, \text{department_id}) \\ \text{department_id} &\rightarrow (\text{department_name}, \text{office_location}) \\ \text{project_id} &\rightarrow \text{project_name} \\ (\text{employee_id}, \text{project_id}) &\rightarrow \text{hours_worked} \end{aligned}$$

The full composite key determines all attributes in the relation. For example, the key determines the employee ID and project ID, which determine the employee and project details; the employee ID also determines the department ID, which determines the department details.

However, the fact that the full key determines every attribute does not mean that the relation is well normalized. We must also examine dependencies on parts of the key and dependencies between non-key attributes.

7.3 Step 2: Check First Normal Form

Suppose every row stores one employee-project assignment, with one value in each attribute. No column contains a list of projects, a list of employees, or a repeating group.

Under these assumptions, the relation is in 1NF.

If instead the company stored a list of project IDs in one employee row, it would not follow the intended relational structure. We would first need to represent each employee-project assignment as its own row.

7.4 Step 3: Check Second Normal Form

The candidate key is composite:

$$K = (\text{employee_id}, \text{project_id})$$

Now examine the dependencies:

$$\text{employee_id} \rightarrow \text{employee_name}$$

The employee name depends only on the employee ID, not on the whole employee-project key.

Similarly,

$\text{project_id} \rightarrow \text{project_name}$

The project name depends only on the project ID.

Both are partial dependencies involving non-prime attributes. Therefore, the original relation violates 2NF.

The same issue applies to `department_id`, which is determined by `employee_id`, and to the department attributes, which are also determined through the employee's department.

To remove the partial dependencies, separate the facts that depend only on an employee, the facts that depend only on a project, and the facts that depend on the complete assignment key.

A first decomposition is:

```
EmployeeAssignmentDetails(  
    employee_id,  
    employee_name,  
    department_id,  
    department_name,  
    office_location  
)  
  
ProjectDetails(  
    project_id,  
    project_name  
)  
  
Assignments(  
    employee_id,  
    project_id,  
    hours_worked  
)
```

The first relation stores employee-related facts. The second stores project-related facts. The third stores the relationship between an employee and a project, along with the hours worked.

In `Assignments`, the hours worked depend on the entire composite key. There are no partial dependencies of non-key attributes on a proper subset of that key.

In `EmployeeAssignmentDetails`, the candidate key is `employee_id`, which is a single attribute. In `ProjectDetails`, the candidate key is `project_id`, also a single attribute. Thus these relations have no partial dependencies on a composite candidate key.

All three relations are in 2NF under the stated assumptions. However, one problem remains.

7.5 Step 4: Check Third Normal Form

Examine the employee-related relation:

```
EmployeeAssignmentDetails(  
    employee_id,  
    employee_name,  
    department_id,  
    department_name,  
    office_location  
)
```

The functional dependencies include:

$$\begin{aligned}\text{employee_id} &\rightarrow (\text{employee_name}, \text{department_id}) \\ \text{department_id} &\rightarrow (\text{department_name}, \text{office_location})\end{aligned}$$

The employee ID determines the department ID, and the department ID determines the department name and office location. Therefore, the employee ID determines those department attributes through the department ID.

The department's name and office location are facts about the department, not about the employee. If ten employees work in the same department, these facts are repeated ten times.

The determinant `department_id` is not a superkey of `EmployeeAssignmentDetails`: it identifies a department, not a single employee. The department name and office location are non-prime attributes. Therefore, these dependencies violate 3NF.

7.6 Step 5: Decompose into Third Normal Form

Move the department information into a separate relation:

```
Employees(  
    employee_id,  
    employee_name,  
    department_id  
)  
  
Departments(  
    department_id,  
    department_name,  
    office_location  
)  
  
Projects(  
    project_id,  
    project_name
```

```
)  
  
Assignments(  
    employee_id,  
    project_id,  
    hours_worked  
)
```

The candidate keys are:

- Employees: employee_id
- Departments: department_id
- Projects: project_id
- Assignments: (employee_id, project_id)

The foreign keys are:

- Employees.department_id references Departments.department_id.
- Assignments.employee_id references Employees.employee_id.
- Assignments.project_id references Projects.project_id.

The employee's department is recorded through the department ID. The department name and office location are stored once per department. Project names are stored once per project. Hours worked remain in the assignment relation because they depend on the employee-project pair.

The resulting design is in 3NF under the stated dependencies. It avoids the original partial dependencies and the transitive dependency that caused department facts to be repeated in the employee relation.

7.7 Step 6: Reconstruct the Original Information

Normalization does not mean that the application can no longer retrieve the original information. It can use joins:

```
SELECT  
    e.employee_id,  
    e.employee_name,  
    d.department_name,  
    d.office_location,  
    p.project_id,
```

```
p.project_name,  
a.hours_worked  
FROM assignments AS a  
JOIN employees AS e  
    ON a.employee_id = e.employee_id  
JOIN departments AS d  
    ON e.department_id = d.department_id  
JOIN projects AS p  
    ON a.project_id = p.project_id;
```

This query reconstructs employee, department, project, and assignment information. The difference is that each fact is maintained in its appropriate relation rather than being repeated in every assignment row.

The normalized design may require joins for certain queries. That is a deliberate trade-off: joins reconstruct related information, while the stored data avoids many update, insertion, and deletion anomalies.

8 How to Determine the Normal Form of a Relation

When given a relation and a set of functional dependencies, use a systematic procedure rather than guessing from the table's appearance.

8.1 Step 1: Determine Candidate Keys

Find which attribute sets determine every attribute in the relation. Then remove unnecessary attributes to identify the minimal such sets. Each minimal superkey is a candidate key.

For example, consider

$$R(A, B, C, D)$$

with functional dependencies

$$A \rightarrow B, \quad B \rightarrow C, \quad C \rightarrow D$$

Starting from A , the dependencies let us determine B , then C , then D . Therefore,

$$A^+ = \{A, B, C, D\}$$

where A^+ is the attribute closure of A . Since A determines every attribute, A is a candidate key, assuming there are no additional attributes or dependencies that change the key analysis.

Because none of the dependencies determines A , every key must include A . The single-attribute set $\{A\}$ is therefore the only candidate key.

8.2 Step 2: Check 1NF

Check whether the relation uses atomic values under the intended relational model and has no repeating groups. If not, redesign the structure before considering the higher normal forms.

8.3 Step 3: Check 2NF

Check every candidate key, especially any composite key. Determine whether a non-prime attribute depends on a proper subset of a candidate key.

If a candidate key is composite and a non-prime attribute depends on only one part of it, the relation violates 2NF.

If all candidate keys are single-attribute keys, the relation is automatically in 2NF once it is in 1NF.

8.4 Step 4: Check 3NF

For every nontrivial functional dependency $X \rightarrow A$, check whether X is a superkey or A is a prime attribute. If neither condition holds, the relation violates 3NF.

A useful initial warning sign is a dependency from a non-key attribute to another non-key attribute. Such dependencies often identify facts that should be moved to a separate relation. However, the formal 3NF definition must be used when multiple candidate keys or prime attributes are involved.

8.5 Step 5: Check BCNF

Boyce–Codd normal form (BCNF) requires every determinant of a nontrivial functional dependency to be a superkey. It is stricter than 3NF.

A relation can satisfy 3NF but violate BCNF, so completing a 3NF decomposition is not necessarily the end of normalization.

9 Attribute Closure and Candidate-Key Calculation

Attribute closure is a systematic way to determine which attributes follow from a given set of attributes under a set of functional dependencies.

Suppose:

$$R(A, B, C, D, E)$$

and the dependencies are:

$$A \rightarrow B, \quad A \rightarrow C, \quad C \rightarrow D, \quad D \rightarrow E$$

Start with A :

$$A^+ = \{A\}$$

Because $A \rightarrow B$ and $A \rightarrow C$, add B and C :

$$A^+ = \{A, B, C\}$$

Because $C \rightarrow D$, add D :

$$A^+ = \{A, B, C, D\}$$

Because $D \rightarrow E$, add E :

$$A^+ = \{A, B, C, D, E\}$$

The closure contains every attribute of the relation, so A is a superkey. It is minimal because the only proper subset of the single-attribute set is the empty set, which does not determine all attributes. Thus A is a candidate key.

In general, to find candidate keys, test attribute sets for whether their closure contains all relation attributes, then verify minimality. A set that determines everything is not necessarily a candidate key if some of its attributes can be removed without losing that property.

One useful shortcut is to inspect attributes that never appear on the right-hand side of any functional dependency. Under the given dependencies, such attributes cannot be derived from other attributes and must belong to every candidate key. This helps narrow the search, but the closure and minimality checks are still necessary.

10 A Second Worked Example: A Relation That Violates 3NF

Consider:

$$R(A, B, C)$$

with the functional dependencies

$$AB \rightarrow C, \quad C \rightarrow B$$

The relation has two candidate keys.

First, AB determines C , so AB determines all attributes. Neither A nor B alone is sufficient, so AB is a candidate key.

Second, AC determines B through $C \rightarrow B$. Therefore AC determines all attributes. Neither A nor C alone is sufficient, so AC is also a candidate key.

The prime attributes are A , B , and C , because each belongs to at least one candidate key. There are no non-prime attributes.

The dependency $C \rightarrow B$ has a determinant that is not a superkey, because C alone does not determine A . Nevertheless, its right-hand-side attribute B is prime. Therefore, this relation satisfies 3NF.

It violates BCNF because C is not a superkey and $C \rightarrow B$ is a nontrivial functional dependency.

This example shows why the informal rule “every non-key attribute depends on the key” is useful for intuition but is not a complete definition of 3NF. Multiple candidate keys and prime attributes can produce cases that require the formal definition.

11 Boyce–Codd Normal Form (BCNF)

11.1 The Basic Idea

BCNF strengthens 3NF by requiring that every determinant in every nontrivial functional dependency be a superkey.

Formally, for every nontrivial functional dependency

$$X \rightarrow Y$$

the relation must have X as a superkey.

The intuition is that if a set of attributes determines another attribute, then that determinant should identify the relevant row or rows uniquely. If it does not, the dependency may cause facts to be repeated across multiple rows.

BCNF is stricter than 3NF. Every relation in BCNF is in 3NF, but some relations in 3NF are not in BCNF.

11.2 A Familiar Example: Courses and Instructors

Suppose a university has the following rule:

Each instructor teaches exactly one course, but a course may be taught by several instructors.

Consider the relation:

<code>Teaching(student_id, course_id, instructor_id)</code>

Assume students may take a course with multiple instructors, and each row records one student-course-instructor combination. The relevant dependencies are:

$$\begin{aligned}(\text{student_id}, \text{instructor_id}) &\rightarrow \text{course_id} \\ \text{instructor_id} &\rightarrow \text{course_id}\end{aligned}$$

The first dependency says that a student's instructor assignment identifies the course. The second follows from the business rule that each instructor teaches exactly one course.

Under these assumptions, the candidate key is $(\text{student_id}, \text{instructor_id})$. The instructor ID alone does not determine the student ID, because an instructor can teach many students.

The dependency

$$\text{instructor_id} \rightarrow \text{course_id}$$

has a determinant that is not a superkey. Therefore, the relation violates BCNF.

However, **course_id** is a prime attribute in this example only if it belongs to some candidate key. Under the stated dependencies, it does not. Thus the relation also violates 3NF.

This example illustrates how a real-world business rule can create dependencies that are not obvious from the table's primary key alone. BCNF requires checking every relevant functional dependency, not just dependencies whose determinants are primary keys.

11.3 Decomposing into BCNF

The violating dependency is

$$\text{instructor_id} \rightarrow \text{course_id}$$

We can separate the instructor-course fact from the student assignment fact:

```
InstructorCourses(instructor_id, course_id)

StudentInstructors(student_id, instructor_id)
```

The first relation records which course an instructor teaches. The second records which instructors a student takes classes with. The course ID is no longer repeated for every student taught by the same instructor.

Under the stated functional dependencies, the two relations satisfy BCNF. The original information can be reconstructed by joining them on `instructor_id`.

The example depends on the stated business rules. If instructors can teach multiple courses, or if students can attend multiple sections of the same course with the same instructor, the functional dependencies and candidate keys must be reconsidered. Normalization conclusions are only as valid as the assumptions on which they are based.

12 Fourth Normal Form (4NF)

Fourth normal form addresses a different kind of redundancy involving multivalued dependencies. It is useful when two independent multivalued facts about the same entity are stored in one relation.

Suppose a person can have several hobbies and speak several languages, and hobbies are independent of languages. Consider:

```
PersonInfo(person_id, hobby, language)
```

If Alex has hobbies of chess and swimming, and speaks English and Spanish, the relation may contain:

person_id	hobby	language
1	Chess	English
1	Chess	Spanish
1	Swimming	English
1	Swimming	Spanish

The four combinations are present because the hobbies and languages are independent lists. The database is repeating each hobby for every language and each language for every hobby.

The design is better represented by two relations:

<code>PersonHobbies(person_id, hobby)</code>
--

<code>PersonLanguages(person_id, language)</code>

The two independent facts can now be stored without generating every combination. If a person acquires a new hobby, the database inserts one hobby row rather than inserting one row for every language the person speaks.

Formally, 4NF requires that for every nontrivial multivalued dependency $X \twoheadrightarrow Y$, X be a superkey. A multivalued dependency is not the same thing as an ordinary functional dependency. It expresses that the set of possible Y values is independent of the remaining attributes, given X .

Fourth normal form is especially useful when a relation combines independent many-valued properties of the same entity.

13 Fifth Normal Form (5NF)

Fifth normal form, also called project-join normal form, addresses certain cases in which a relation can be decomposed into several smaller relations and reconstructed by joins without introducing spurious rows.

A relation is in 5NF when every nontrivial join dependency is implied by its candidate keys. Informally, there should be no additional nontrivial decomposition into smaller relations that preserves the intended information and is required by the join dependencies.

A common teaching example involves suppliers, parts, and projects. Suppose a business records which suppliers can supply which parts, which suppliers are approved for which projects, and which parts are used by which projects. Under particular business rules, the valid supplier-part-project combinations may be reconstructed by joining three pairwise relations.

If the business rules genuinely imply that a triple is valid exactly when all three corresponding pairs are valid, storing the triples directly may create redundancy. However, that conclusion cannot be drawn merely because the data involves three entities. The decomposition is correct only if the join dependency follows from the business rules.

Fifth normal form is less common in introductory database design than 1NF, 2NF, and 3NF. The important lesson is that some redundancy arises from relationships among several projections of a relation, even when simpler functional dependency analysis does not reveal the problem.

14 Lossless Decomposition and Dependency Preservation

Normalization decomposes a relation into smaller relations. A good decomposition should preserve the original information and, where possible, make the important functional dependencies easy to enforce.

14.1 Lossless Decomposition

A decomposition is lossless when joining the decomposed relations reconstructs exactly the original relation, without losing information or generating invalid extra rows.

For example, suppose we decompose:

```
Employees(employee_id, employee_name, department_id,  
           department_name)
```

into:

```
Employees(employee_id, employee_name, department_id)  
  
Departments(department_id, department_name)
```

If `department_id` determines `department_name`, the decomposition can be lossless under the stated constraints. The department ID provides the link needed to reconstruct each employee's department name.

A decomposition that is not lossless may create spurious rows when the tables are joined. That would be a serious error: the reconstructed relation could contain combinations that were never valid in the original data.

14.2 Dependency Preservation

A decomposition is dependency-preserving if the original functional dependencies can be enforced by checking the decomposed relations individually, without needing to join them all together.

For example, if the employee ID determines the employee name, placing both attributes in the **Employees** relation makes that dependency straightforward to enforce and reason about. Similarly, storing department ID and department name together in **Departments** keeps the department dependency local to that relation.

Dependency preservation is desirable because checking constraints within a single relation is often simpler and more efficient than checking constraints that require joins.

Losslessness and dependency preservation are different properties. A decomposition can be lossless without preserving every functional dependency locally. In particular, BCNF decomposition may require a trade-off in which not every original dependency can be enforced by inspecting one decomposed relation alone.

15 Normalization and Practical Database Design

Normalization is a logical design technique, not an instruction to split every table into as many pieces as possible. A well-normalized design reduces redundancy and prevents anomalies while preserving the information and constraints required by the application.

Normalized tables often require joins to retrieve related facts. This is not inherently a disadvantage. Relational databases are designed to join relations, and indexes and query optimization help execute many joins efficiently. The benefit is that each fact has a clear location and can be updated consistently.

In some analytical systems, data warehouses, reporting layers, or performance-sensitive applications, designers intentionally use denormalized structures. For example, a reporting table might store a department name alongside every employee record to simplify queries. Such redundancy can be justified when the trade-offs are understood and the data is maintained consistently.

Denormalization should be an intentional design choice rather than an accidental result of failing to identify functional dependencies. If the same fact is stored in multiple places, the system must have a reliable strategy for keeping those copies synchronized.

16 Summary

Normalization organizes relational data according to the dependencies among its attributes. Candidate keys identify rows, while functional dependencies express which attributes determine other attributes. These concepts provide the basis for deciding whether a relation satisfies a normal form.

The most important introductory normal forms are:

- **1NF:** Values are atomic in the intended relational model; repeating groups are eliminated.
- **2NF:** The relation is in 1NF, and no non-prime attribute depends on only part of a candidate key.
- **3NF:** The relation is in 2NF, and every nontrivial functional dependency has a superkey as its determinant or a prime attribute on its right-hand side.

- **BCNF:** Every determinant of a nontrivial functional dependency is a superkey.
- **4NF:** Nontrivial multivalued dependencies must have a superkey as their determinant.
- **5NF:** Nontrivial join dependencies must be implied by candidate keys.

In practical terms, normalization asks three important questions:

1. What uniquely identifies each row?
2. What facts depend on that identifier, or on parts of that identifier?
3. Are facts about one entity being unnecessarily repeated in rows that primarily describe another entity or relationship?

The goal is to store each fact in an appropriate place, preserve the relationships among facts, and reduce the possibility of inconsistent updates. Functional dependencies and candidate keys turn this goal into a systematic method for designing relational schemas.