Normal form and normalization

• A **normal form** is a property of a relational database.

• When a relation is *non-normalized* (that is, does not satisfy a normal form), then it presents redundancies and produces undesirable behaviour during update operations.

• This principle can be used to carry out quality analysis and constitutes a useful tool for database design.

• **Normalization** is a procedure that allows the non-normalized schemas to be transformed into new schemas for which the satisfaction of a normal form is guaranteed.

Example of a relation with anomalies

Employee	Salary	Project	Budget	Function
Brown	20	Mars	2	technician
Green	35	Jupiter	15	designer
Green	35	Venus	15	designer
Hoskins	55	Venus	15	manager
Hoskins	55	Jupiter	15	consultant
Hoskins	55	Mars	2	consultant
Moore	48	Mars	2	manager
Moore	48	Venus	15	designer
Kemp	48	Venus	15	designer
Kemp	48	Jupiter	15	manager

The key is composed of the attributes Employee and Project

Anomalies in the example relation

• The value of the salary of each employee is repeated in all the tuples relating to it: therefore there is a **redundancy**.

• If the salary of an employee changes, we have to modify the value in all the corresponding tuples. This problem is known as the **update anomaly**.

• If an employee stops working on all the projects but does not leave the company, all the corresponding tuples are deleted (note that Project is part of the key) and so, even the basic information, name and salary is lost. This problem is known as the **deletion anomaly**.

• If we have information on a new employee, we cannot insert it until the employee is assigned to a project. This is known as the **insertion anomaly**.

Why these undesirable phenomena?

• Intuitive explanation: we have used a single relation to represent items of information of different types.

• In particular, the following independent real-world concepts are represented in the relation:

– employees with their salaries,

– projects with their budgets,

– participation of the employees in the projects with their functions.

• To formally study these principles introduced informally, it is necessary to define the notion of **functional dependency**.

Functional dependencies

• Given a relation *R* on a schema *R(X)* and two non-empty subsets *Y* and *Z* of its attributes X, we say that there is a **functional dependency** on *R* between *Y* and Z, if, for each pair of tuples *t1* and *t2* of *R* having the same values for the attributes in Y, *t1* and *t2* also have the same values for the attributes in *Z*.

• A functional dependency between the sets of attributes *Y* and *Z* is indicated by the notation $Y \rightarrow Z$

Functional dependencies in the example schema

• Employee → Salary

the salary of each employee is unique and thus each time a certain employee appears in a tuple, the value of his or her salary always remains the same.

• Project → Budget

the budget of each project is unique and thus each time a certain project appears in a tuple, the value of its budget always remains the same.

Non-trivial functional dependencies

• We say that a functional dependency $Y \rightarrow Z$ is **non-trivial** if no attribute in Z appears among the attributes in Y.

– Employee → Salary is a non-trivial functional dependency

– Employee Project → Project is a trivial functional dependency

Anomalies and functional dependencies

In our example, the two properties causing anomalies correspond exactly to attributes involved in functional dependencies:

– the property “the salary of each employee is unique and depends only on the employee” corresponds to the functional dependency

Employee → Salary

– the property “the budget of each project is unique and depends only on the project” corresponds to the functional dependency

Project → Budget

Moreover, the following property can be formalized by means of a functional dependency:

– the property “for each project, each of the employees involved can carry out only one function” corresponds to the functional dependency

Employee, Project → Function

Dependencies generating anomalies

• The first two dependencies generate undesirable redundancies and anomalies.

• On the contrary, the third dependency

Employee, Project → Function

never generates redundancies because, having {Employee, Project} as a key, the relation cannot contain two different tuples with the same values for these attributes.

• The difference between the first two dependencies and the third is that Employee and Project together (i.e., the left hand side of the third functional dependency) form the key of the relation but individually they are not keys.

Boyce–Codd Normal Form (BCNF)

• A relation *R* is in **Boyce–Codd normal form** if

- for every (non-trivial) functional dependency $X \rightarrow Y$ defined on it, *X* contains a key *K* of *R*.

• Anomalies and redundancies, as discussed above, do not appear in databases with relations in Boyce–Codd normal form, because the independent pieces of information are separate, one per relation.

Decomposition into Boyce–Codd normal form

• Given a relation that does not satisfy Boyce–Codd normal form, we can often replace it with one or more normalized relations using a process called **normalization**.

• For instance, we can eliminate redundancies and anomalies for the relation of the previous example if we replace it with (**decompose** it into) the three relations, obtained by projections on the sets of attributes corresponding to the three functional dependencies.

• In this case, the key of each of the relations we obtain are the left hand side of the corresponding functional dependency: the satisfaction of the Boyce–Codd normal form is therefore guaranteed.

Decomposition of the example relation

Employee	Salary
Brown	20
Green	35
Hoskins	55
Moore	48
Kemp	48

Project	Budget
Mars	2
Jupiter	15
Venus	15

Employee	Project	Function
Brown	Mars	technician
Green	Jupiter	designer
Green	Venus	designer
Hoskins	Venus	manager
Hoskins	Jupiter	consultant
Hoskins	Mars	consultant
Moore	Mars	manager
Moore	Venus	designer
Kemp	Venus	designer
Kemp	Jupiter	manager

A relation to be decomposed

Employee	Project	Branch
Brown	Mars	Chicago
Green	Jupiter	Birmingham
Green	Venus	Birmingham
Hoskins	Saturn	Birmingham
Hoskins	Venus	Birmingham

The relation satisfies the functional dependencies:

- Employee → Branch
- Project → Branch

The key of the relation is composed of Employee and Project

A possible decomposition of the previous relation

Employee	Branch
Brown	Chicago
Green	Birmingham
Hoskins	Birmingham

Project	Branch
Mars	Chicago
Jupiter	Birmingham
Saturn	Birmingham
Venus	Birmingham

The join of the projections

Employee	Project	Branch
Brown	Mars	Chicago
Green	Jupiter	Birmingham
Green	Venus	Birmingham
Hoskins	Saturn	Birmingham
Hoskins	Venus	Birmingham
Green	Saturn	Birmingham
Hoskins	Jupiter	Birmingham

The result is different from the original relation: the information can not be reconstructed.

Lossless decomposition

- The decomposition of a relation R into two tables R1 and R2 is ***lossless*** if the *join* of R1 and R2 is equal to R (that is, not containing *spurious* tuples)

- It is clearly desirable, or rather an indispensable requirement, that a decomposition carried out for the purpose of normalization is lossless

A lossless decomposition of the previous relation

Employee	Branch
Brown	Chicago
Green	Birmingham
Hoskins	Birmingham

Employee	Project
Brown	Mars
Green	Jupiter
Green	Venus
Hoskins	Saturn
Hoskins	Venus

Another problem with the new decomposition

- Assume we wish to insert a new tuple that specifies the participation of the employee named Armstrong, who works in Birmingham, on the Mars project.

- In the original relation this update would be immediately identified as illegal, because it would cause a violation of the **Project** → **Branch** dependency.

- On the decomposed relations however, it is not possible to reveal any violation of dependency since the two attributes Project and Branch have been separated: one into one relation and one into the other.

Preservation of dependencies

- A decomposition **preserves the dependencies** if each of the functional dependencies of the original schema involves attributes that appear all together in one of the decomposed schemas.

- It is clearly desirable to preserve dependencies when decomposing a table as, in this way, it is possible to ensure, the satisfaction of the same constraints in the decomposed schema, as in the original schema.

Qualities of decompositions

- Decompositions should always satisfy the properties of lossless decomposition and dependency preservation:
  - Lossless decomposition ensures that the information in the original relation can be accurately reconstructed based on the information represented in the decomposed relations.
  - Dependency preservation ensures that the decomposed relations have the same capacity to represent the integrity constraints as the original relations and thus to reveal illegal updates.

A relation not satisfying BCNF

Manager	Project	Branch
Brown	Mars	Chicago
Green	Jupiter	Birmingham
Green	Mars	Birmingham
Hoskins	Saturn	Birmingham
Hoskins	Venus	Birmingham

The key for this relation is {Project, Branch}

Assume that the following dependencies are defined:

- **Manager** → **Branch**: each manager works at a particular branch;
- **Project, Branch** → **Manager**: each project has more managers who are responsible for it, but in different branches, and each manager can be responsible for more than one project; however, for each branch, a project has only one manager responsible for it.

A problematic decomposition

- The key for this relation is {Project, Branch}
- The relation is not in Boyce–Codd normal form because the left hand side of the dependency **Manager** → **Branch** does not contain the entire key.
- At the same time, no good decomposition of this relation is possible: the dependency

  Project, Branch → **Manager**

  involves all the attributes and thus no decomposition is able to preserve it.
- We can therefore state that sometimes, Boyce–Codd normal form cannot be achieved.

A new normal form

A relation *r* is in **third normal form** if, for each (non trivial) functional dependency $X \rightarrow Y$ defined on it, at least one of the following is verified:

– *X* contains a key *K* of *r*;

– each attribute in *Y* is contained in at least one key of *r*.

BCNF and third normal form

- The previous schema does not satisfy Boyce–Codd normal form, but it satisfies the third normal form:

  – The **Project, Branch** → **Manager** dependency has as its left hand side a key for the relation, while **Manager** → **Branch** has a unique attribute on the right hand side, which is part of the {Project, Branch} key.

- Third normal form is less restrictive than Boyce–Codd normal form and for this reason does not offer the same guarantees of quality for a relation; it has the advantage however, of always being achievable.

Decomposition into third normal form

- Decomposition into third normal form can proceed as suggested for the Boyce–Codd normal form:
  - a relation that does not satisfy third normal form can be decomposed into relations obtained by projections on the attributes corresponding to the functional dependencies.

- An important condition to guarantee in this process is of always maintaining a relation that contains a key of the original relation

Database design and normalization

The theory of normalization can be used as a basis for quality control operations on schemas, in both the conceptual and logical design phases:

– the analysis of the relations obtained during the logical design phase can identify places where the conceptual design was inaccurate: this verification of the design is often relatively easy;

– the ideas on which normalization is based can also be used during the conceptual design phase for the quality control of each element of the conceptual schema.