**Basics of Functional**

**Dependencies and Normalization**

**for Relational Databases**

There are two levels at which we can discuss the *goodness* of relation schemas. The

first is the **logical** (or **conceptual**) **level**—how users interpret the relation schemas

and the meaning of their attributes. Having good relation schemas at this level

enables users to clearly understand the meaning of the data in the relations, and

hence to formulate their queries correctly. The second is the **implementation** (or

**physical storage**) **level**—how the tuples in a base relation are stored and updated.

Top down vs. bottom-up design methodology:

In contrast, a **top-down design methodology** (also called *design by*

*analysis*) starts with a number of groupings of attributes into relations that exist

together naturally, for example, on an invoice, a form, or a report. The relations are

then analyzed individually and collectively, leading to further decomposition until all

desirable properties are met.

Relational database design ultimately produces a set of relations. The implicit goals

of the design activity are *information preservation* and *minimum redundancy*.

Information is very hard to quantify—hence we consider information preservation

in terms of maintaining all concepts, including attribute types, entity types, and

relationship types as well as generalization/specialization relationships, which are

described using a model such as the EER model. Thus, the relational design must

preserve all of these concepts, which are originally captured in the conceptual

design after the conceptual to logical design mapping.

Informational guidelines to measure quality of relation schema:

■ 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

**14.1.1 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.

*\*Should be true if we follow the process in chapter 9.*

In general, the easier it is to explain the semantics of the relation—or in other words,

what a relation exactly means and stands for—the better the relation schema design

will be.

Example:

The meaning of the EMPLOYEE relation

schema is simple: Each tuple represents an employee, with values for the employee’s

name (Ename), Social Security number (Ssn), birth date (Bdate), and address

(Address), and the number of the department that the employee works for (Dnumber).





The ease with which the meaning of a relation’s attributes can be explained

is an *informal measure* of how well the relation is designed.

**Guideline 1.** Design a relation schema so that it is easy to explain its meaning. Do

not combine attributes from multiple entity types and relationship types into a single

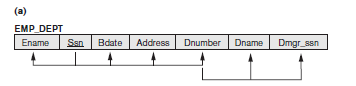
relation. Intuitively, if a relation schema corresponds to one entity type or one

relationship type, it is straightforward to explain its meaning. Otherwise, if the relation

corresponds to a mixture of multiple entities and relationships, semantic ambiguities

will result and the relation cannot be easily explained.

Example violation:



**14.1.2 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 can be classified into insertion anomalies, deletion anomalies,

and modification anomalies.

**Insertion Anomalies.** Insertion anomalies can be differentiated into two types,

illustrated by the following examples based on the EMP\_DEPT relation:

■ To insert a new employee tuple into EMP\_DEPT, we must include either the

attribute values for the department that the employee works for, or NULLs (if

the employee does not work for a department as yet). For example, to insert

a new tuple for an employee who works in department number 5, we must

enter all the attribute values of department 5 correctly so that they are *consistent*

with the corresponding values for department 5 in other tuples in

EMP\_DEPT. In the design of Figure 14.2, we do not have to worry about this

consistency problem because we enter only the department number in the

employee tuple; all other attribute values of department 5 are recorded only

once in the database, as a single tuple in the DEPARTMENT relation.

***\*Can result in inconsistent values for an existing department 5 for department name, and department manager Ssn.***

■ It is difficult to insert a new department that has no employees as yet in the

EMP\_DEPT relation. The only way to do this is to place NULL values in the

attributes for employee. This violates the entity integrity for EMP\_DEPT

because its primary key Ssn cannot be null. Moreover, when the first

employee is assigned to that department, we do not need this tuple with

NULL values anymore. This problem does not occur in the design of Figure

14.2 because a department is entered in the DEPARTMENT relation whether

or not any employees work for it, and whenever an employee is assigned to

that department, a corresponding tuple is inserted in EMPLOYEE.

***\*Must violate entity integrity to insert only a department without an employee assigned as it’s manager. Then when the employee is finally inserted that tuple with Null values becomes irrelevant.***

**Deletion Anomalies.** The problem of deletion anomalies is related to the second

insertion anomaly situation just discussed. 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 inadvertently from the

database. This problem does not occur in the database of Figure 14.2 because

DEPARTMENT tuples are stored separately.

***\*If we delete the last employee for a given department, we will lose that information.***

**Modification Anomalies.** In EMP\_DEPT, if we change the value of one of the attributes

of a particular department—say, the manager of department 5—we must

update the tuples of *all* employees who work in that department; otherwise, the

database will become inconsistent. If we fail to update some tuples, the same department

will be shown to have two different values for manager in different employee

tuples, which would be wrong.3

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, we can state the next guideline as follows.

*\* Updating the name for a department may lead to inconsistencies across other tuples for that same department.*

**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,4

note them clearly and make sure that the programs that update the database will

operate correctly.

It is important to note that these guidelines may sometimes

*have to be violated* in order to *improve the performance* of certain queries.

***\*These potentials for anomalies must be noted and accounted for.***

**14.1.3 NULL Values in Tuples**

In some schema designs we may group many attributes together into a “fat” relation.

If many of the attributes do not apply to all tuples in the relation, we end up

with many NULLs in those tuples.

*Waste space at the storage level, lead to misunderstanding of the meaning of attributes, joins, aggregate functions, and comparisons in queries.*

**Guideline 3.** As far 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 and do not apply to most tuples in the relation.

Using space efficiently and avoiding joins with NULL values are the two overriding

criteria that determine whether to include the columns that may have NULLs in a

relation or to have a separate relation for those columns (with the appropriate key

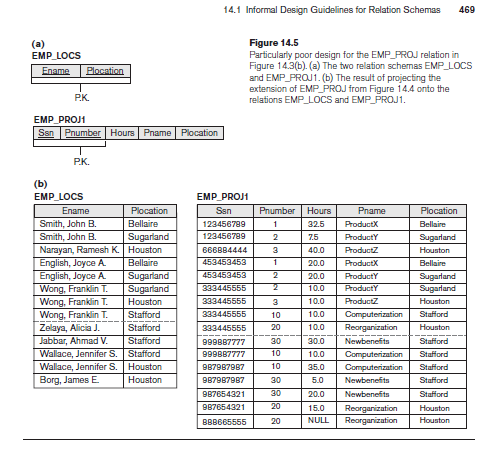
columns). For example, if only 15% of employees have individual offices, there is

little justification for including an attribute Office\_number in the EMPLOYEE relation;

rather, a relation EMP\_OFFICES(Essn, Office\_number) can be created to include

tuples for only the employees with individual offices.

**14.1.4 Generation of Spurious Tuples**



**spurious tuples**

because they represent spurious information that is not valid.

**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:*

■ 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

**14.2 Functional Dependencies**

So far we have dealt with the informal measures of database design. We now introduce

a formal tool for analysis of relational schemas that enables us to detect and

describe some of the above-mentioned problems in precise terms.

**Definition.** A **functional dependency**, denoted by *X ->* *Y*, between two sets of

attributes *X* and *Y* that are subsets of *R* specifies a *constraint* on the possible

tuples that can form a relation state *r* of *R*. The constraint is that, for any two

tuples *t*1 and *t*2 in *r* that have *t*1[*X*] = *t*2[*X*], they must also have *t*1[*Y*] = *t*2[*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; alternatively, the values of the *X* component

of a tuple uniquely (or **functionally**) *determine* the values of the *Y* component. We

also say that there is a functional dependency from *X* to *Y*, or that *Y* is **functionally**

**dependent** on *X*. The abbreviation for functional dependency is **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**.

Thus, *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. Note the following:

■ 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* -> *Y* for any subset of attributes *Y* of *R* (because the

key constraint implies that no two tuples in any legal state *r*(*R*) will have the

same value of *X*). If *X* is a candidate key of *R*, then *X* -> *R.*

■ If *X* -> *Y* in *R*, this does not say whether *Y* -> *X* in *R*.