Relational Model

Unit 3

The Relational Model

Brief History of the Relational Model

The relational model was first proposed by E. F. Codd in his seminal paper “A relational model of data for large shared data banks” (Codd, 1970). This paper is now generally accepted as a landmark in database systems.

The relational model’s objectives were specified as follows:

- To allow a high degree of data independence. Application programs must not be affected by modifications to the internal data representation, particularly by changes to file organizations, record orderings, or access paths.
- To provide substantial grounds for dealing with data semantics, consistency, and redundancy problems. In particular, Codd’s paper introduced the concept of **normalized** relations, that is, relations that have no repeating groups.
- To enable the expansion of set-oriented data manipulation languages.

Relation, Attribute and Domain

Relation - A relation is a table with columns and rows. However, this perception applies only to the logical structure of the database.

Attribute - An attribute is a named column of a relation. Attributes can appear in any order and the relation will still be the same relation.

Domain - A domain is the set of allowable values for one or more attributes. Every attribute in a relation is defined on a domain. Domains may be distinct for each attribute, or two or more attributes may be defined on the same domain.

Properties of Relations

A relation has the following properties:

- the relation has a name that is distinct from all other relation names in the relational schema;
- each cell of the relation contains exactly one atomic (single) value;
- each attribute has a distinct name;
- the values of an attribute are all from the same domain;
- each tuple is distinct; there are no duplicate tuples;
- the order of attributes has no significance;
- the order of tuples has no significance, theoretically. (However, in practice, the order may affect the efficiency of accessing tuples.)

Domain Constraints

- A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types).
- Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take.
- Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.

Integrity Constraints

A data model has two other parts: a manipulative part, defining the types of operation that are allowed on the data, and a set of integrity constraints, which ensure that the data is accurate.

As we know every attribute has an associated domain, there are constraints (called domain constraints) that form restrictions on the set of values allowed for the attributes of relations.

In addition, there are two important integrity rules, which are **constraints** or **restrictions** that apply to all instances of the database. The two principal rules for the relational model are known as **entity integrity** and **referential integrity**.

Entity integrity: In a base relation, no attribute of a primary key can be null.

Integrity Constraints

Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damage to the database.

Examples of integrity constraints are:

• An instructor name cannot be null.

• No two instructors can have the same instructor ID.

• Every department name in the course relation must have a matching department name in the department relation.

• The budget of a department must be greater than $0.00.

Integrity Constraints

In general, an integrity constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, most database systems allow one to specify integrity constraints that can be tested with minimal overhead.

Integrity constraints are usually identified as part of the database schema design process, and declared as part of the **create table** command used to create relations.

However, integrity constraints can also be added to an existing relation by using the command **alter table** *table-name* **add** *constraint*, where *constraint* can be any constraint on the relation.

When such a command is executed, the system first ensures that the relation satisfies the specified constraint. If it does, the constraint is added to the relation; if not, the command is rejected.

Referential Integrity

When, we wish to ensure that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation. This condition is called **referential integrity**.

For example,

The department listed for each course must be one that actually exists. More precisely, the *dept_name* value in a course record must appear in the *dept_name* attribute of some record of the *department* relation.

Database modifications can cause violations of referential integrity.When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.

Referential Integrity

More generally, let *r1* and *r2* be relations whose set of attributes are *R1* and *R2*, respectively, with primary keys *K1* and *K2*.

We say that, a subset α of *R2* is a foreign key referencing *K1* in relation *r1* if it is required that, for every tuple *t2* in *r2*, there must be a tuple *t1* in *r1* such that

$$t1.K1 = t2.\alpha$$

Requirements of this form are called **referential-integrity constraints**, or **subset dependencies**.

Referential Integrity

Note that, for a referential-integrity constraint to make sense, ***α*** and *K1* must be compatible sets of attributes; that is,either must be equal to *K1*, or they must contain the same number of attributes, and the types of corresponding attributes must be compatible (we assume here that and *K1* are ordered).

In general, a referential integrity constraint does not require *K1* to be a primary key of *r1*; as a result,more than one tuple in *r1* can have the same value for attributes *K1*.

Referential Integrity

By default, in SQL a foreign key references the primary-key attributes of the referenced table.

SQL also supports a version of the **references** clause where a list of attributes of the referenced relation can be specified explicitly. The specified list of attributes must, however, be declared as a candidate key of the referenced relation, using either a **primary key** constraint, or a **unique** constraint.

We can use the following short form as part of an attribute definition to declare that the attribute forms a foreign key:

> *dept_name* **varchar**(20) **references** *department*

When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation (that is, the transaction performing the update action is rolled back).

Referential Integrity

However, a **foreign key** clause can specify that if a delete or update action on the referenced relation violates the constraint, then, instead of rejecting the action, the system must take steps to change the tuple in the referencing relation to restore the constraint.

Consider this definition of an integrity constraint on the relation course:

```
create table course
    ( . . .
    foreign key (dept_name) references department
        on delete cascade
        on update cascade,
    . . . );
```

Because of the clause **on delete cascade** associated with the foreign-key declaration, if a delete of a tuple in *department* results in this referential-integrity constraint being violated, the system does not reject the delete. Instead, the delete "cascades" to the *course* relation, deleting the tuple that refers to the department that was deleted.

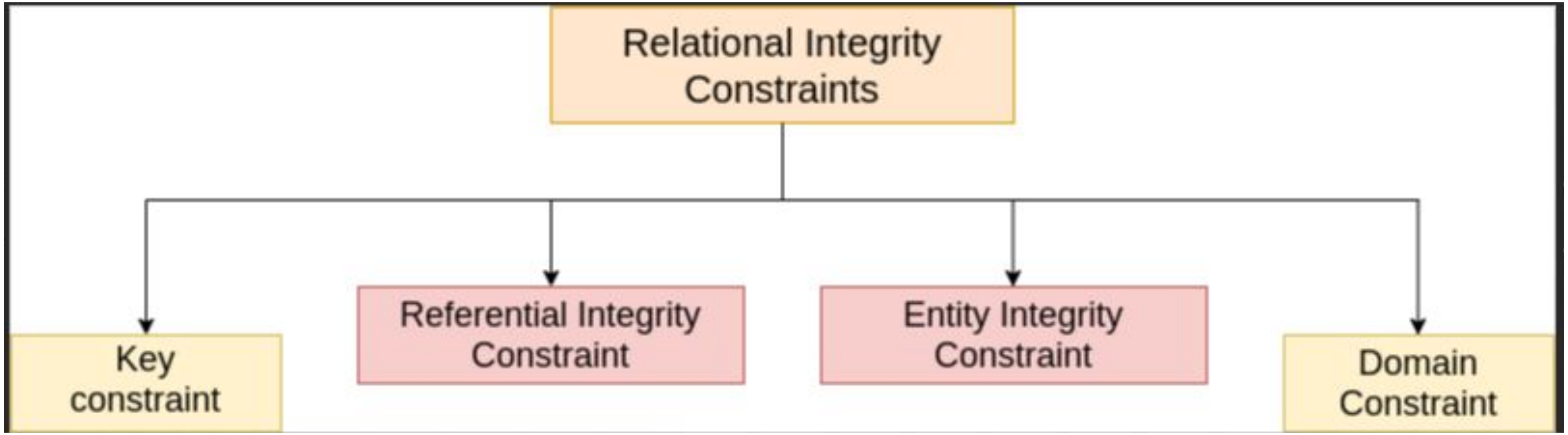
Referential Integrity

SQL also allows the **foreign key** clause to specify actions other than **cascade**, if the constraint is violated: The referencing field (here, *dept_name*) can be set to *null* (by using set **null** in place of cascade), or to the default value for the domain (by using **set default**).

Null values complicate the semantics of referential-integrity constraints in SQL. Attributes of foreign keys are allowed to be *null*, provided that they have not otherwise been declared to be **not null**.

If all the columns of a foreign key are nonnull in a given tuple, the usual definition of foreign-key constraints is used for that tuple. If any of the foreign-key columns is *null*, the tuple is defined automatically to satisfy the constraint.

Relational Integrity Constraints


Relational Integrity Constraints
Key constraint
Referential Integrity Constraint
Entity Integrity Constraint
Domain Constraint

Database Design

Relational Model

- The goal of relational database design is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. This is accomplished by designing schemas that are in an appropriate normal form.
- To determine whether a relation schema is in one of the desirable normal forms, we need information about the real-world enterprise that we are modeling with the database.
- Some of this information exists in a well-designed E-R diagram, but additional information about the enterprise may be needed as well.

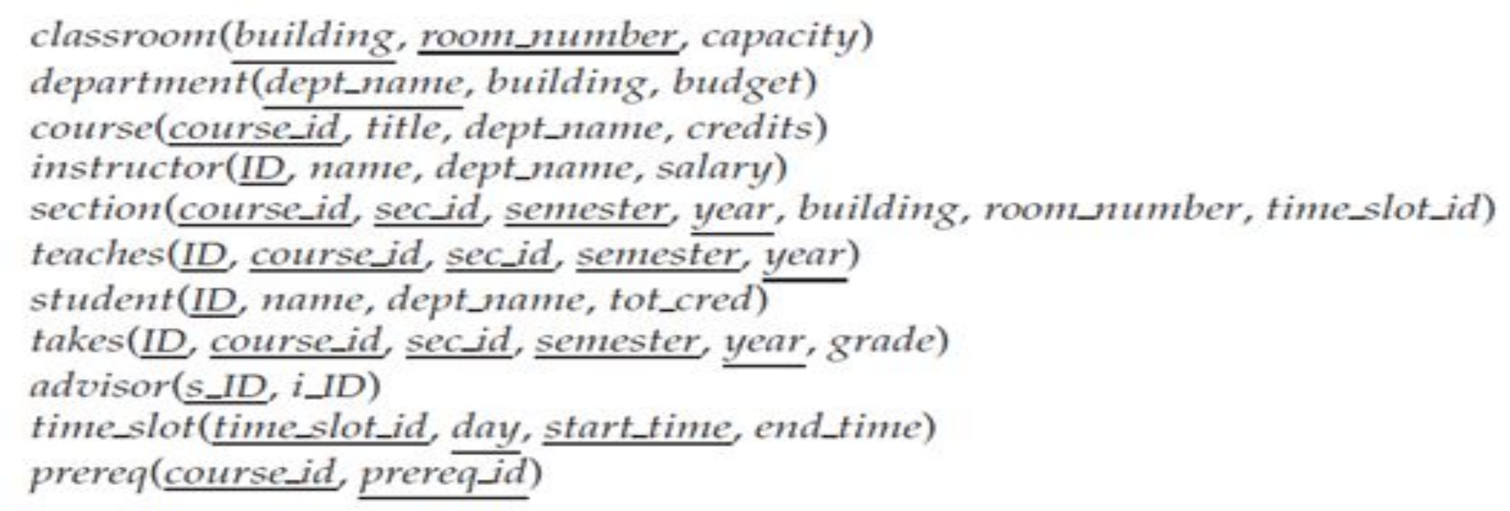
Relational Model

Here we discuss a formal approach to relational database design based on the notion of functional dependencies.

We then define normal forms in terms of functional dependencies and other types of data dependencies.

Features of Good Relational Designs

The goodness (or badness) of the resulting set of schemas depends on how good the E-R design was in the first place.



classroom(building, room_number, capacity)
department(dept_name, building, budget)
course(course_id, title, dept_name, credits)
instructor(ID, name, dept_name, salary)
section(course_id, sec_id, semester, year, building, room_number, time_slot_id)
teaches(ID, course_id, sec_id, semester, year)
student(ID, name, dept_name, tot_cred)
takes(ID, course_id, sec_id, semester, year, grade)
advisor(s_ID, i_ID)
time_slot(time_slot_id, day, start_time, end_time)
prereq(course_id, prereq_id)

Figure 8.1 Schema for the university database.

Design Alternative: Larger Schemas

Let us explore features of this relational database design as well as some alternatives. Suppose that instead of having the schemas instructor and department, we have the schema:

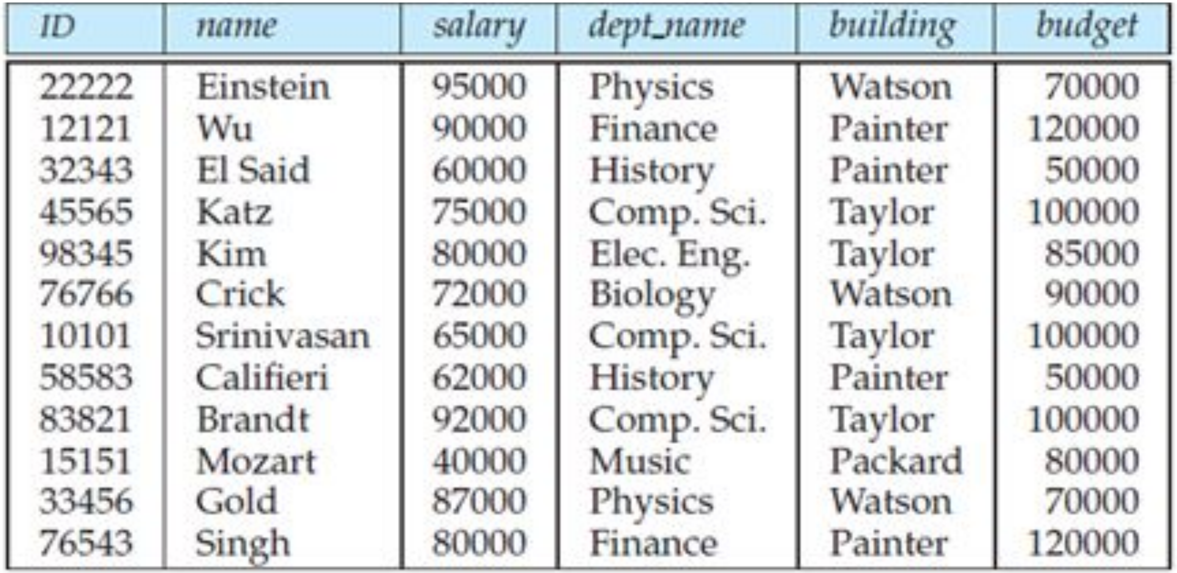
inst_dept (ID, name, salary, dept_name, building, budget)

This represents the result of a natural join on the relations corresponding to *instructor* and *department*.

This seems like a good idea because some queries can be expressed using fewer joins, until we think carefully about the facts about the university that led to our E-R design.

Design Alternative: Larger Schemas

Let us consider the instance of the *inst_dept* relation shown in Figure



ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Figure 8.2 The *inst_dept* table.

Design Alternative: Larger Schemas

In the above table, notice that we have to repeat the department information ("building" and "budget") once for each instructor in the department. For example, the information about the Comp. Sci. department (Taylor, 100000) is included in the tuples of instructors Katz, Srinivasan, and Brandt. It is important that all these tuples agree as to the budget amount since otherwise our database would be inconsistent.

In our original design using *instructor* and *department*, we stored the amount of each budget exactly once.

This suggests that using *inst_dept* is a bad idea since it stores the budget amounts redundantly and runs the risk that some user might update the budget amount in one tuple but not all, and thus create inconsistency.

Design Alternative: Larger Schemas

Even if we decided to live with the redundancy problem, there is still another problem with the *inst_dept* schema.

Suppose we are creating a new department in the university.

In the alternative design above, we cannot represent directly the information concerning a department (dept name, building, budget) unless that department has at least one instructor at the university. This is because tuples in the *inst_dept* table require values for *ID, name, and salary*. This means that we cannot record information about the newly created department until the first instructor is hired for the new department.

In the old design, the schema department can handle this, but under the revised design, we would have to create a tuple with a null value for building and budget. In some cases null values are also troublesome.

Design Alternative: Smaller Schemas

In the case of *inst_dept*,we need to allow the database designer to specify rules such as "each specific value for *dept_name* corresponds to atmost one *budget*" even in cases where dept_name is not the primary key for the schema in question.

In other words, we need to write a rule that says "if there were a schema (*dept_name*, *budget*), then *dept_name* is able to serve as the primary key." This rule is specified as a functional dependency

$$dept\_name \rightarrow budget$$

Observations such as these and the rules (functional dependencies in particular) that result from them allow the database designer to recognize situations where a schema ought to be **split**, or **decomposed**, into two or more schemas.

Design Alternative: Smaller Schemas

The database designer had to recognize situations where a schema ought to be split, or decomposed, into two or more schemas considering the rules and functional dependencies. Finding the right decomposition is much harder for schemas with a large number of attributes and several functional dependencies.

Consider a less extreme case where we choose to decompose the employee schema

employee (ID, name, street, city, salary)

into the following two schemas:

employee1 (ID, name)

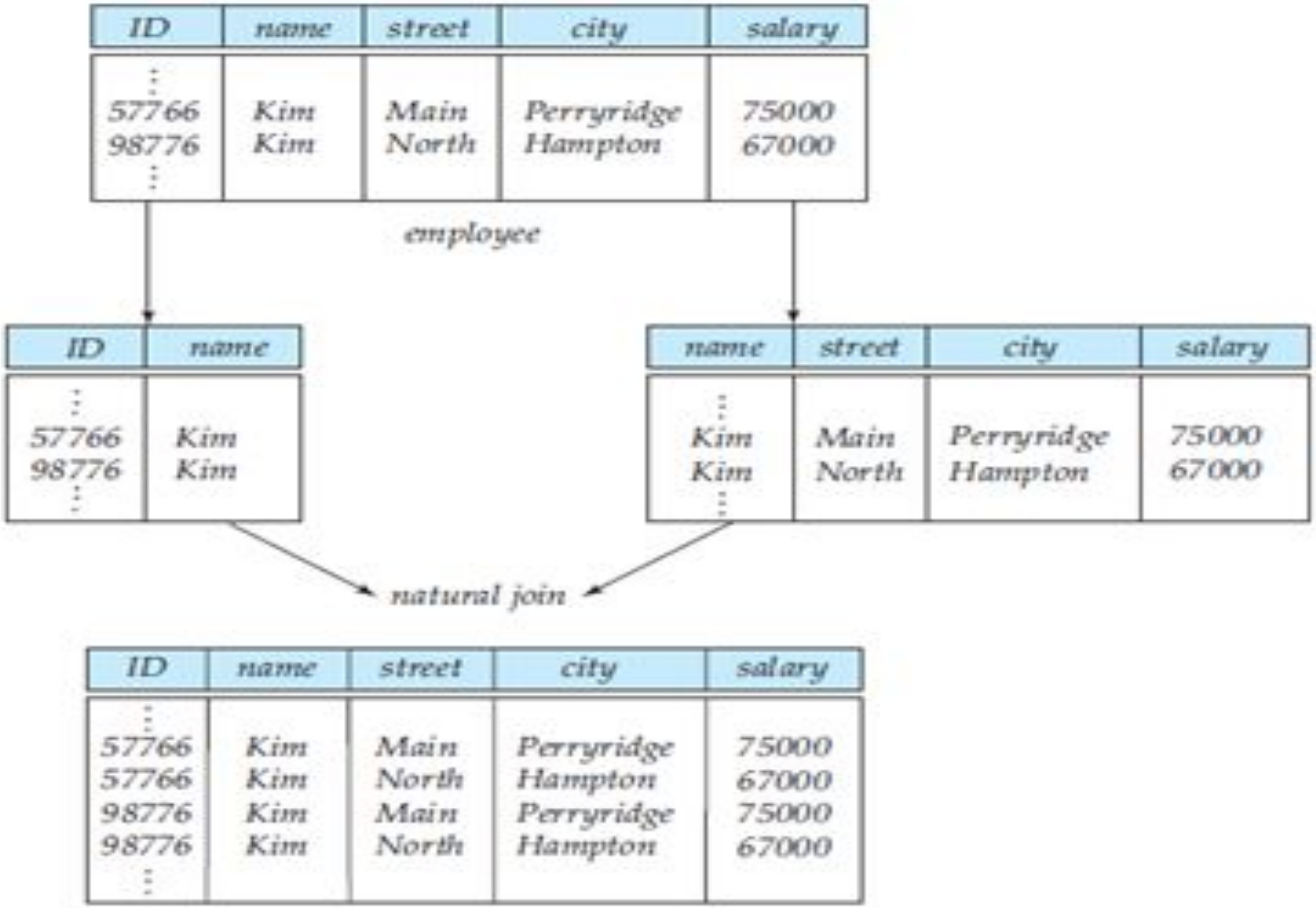
employee2 (name, street, city, salary)

The flaw in this decomposition arises from the possibility that the enterprise has two employees with the same name.

For example, let us assume two employees, both named Kim, work at the university and have the following tuples in the relation on schema employee in the original design:

(57766, Kim, Main, Perryridge, 75000)

(98776, Kim, North, Hampton, 67000)



ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

ID	name
⋮	
57766	Kim
98776	Kim
⋮	

name	street	city	salary
⋮			
Kim	Main	Perryridge	75000
Kim	North	Hampton	67000
⋮			

ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
57766	Kim	North	Hampton	67000
98776	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

Figure 8.3 Loss of information via a bad decomposition.

Design Alternative: Smaller Schemas

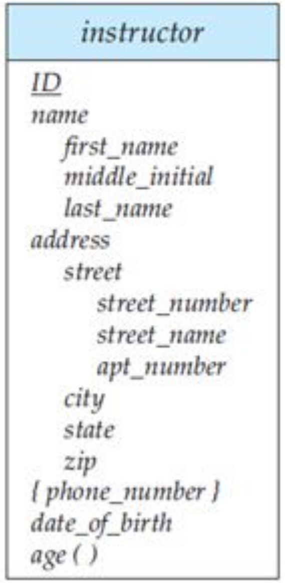
As we see in the figure, the two original tuples appear in the result along with two new tuples that incorrectly mix data values pertaining to the two employees named Kim.

Thus, our decomposition is unable to represent certain important facts about the university employees. Clearly, we would like to avoid such decompositions.

We shall refer to such decompositions as being **lossy decompositions**.

Atomic Domains and First Normal Form

- The E-R model allows entity sets and relationship sets to have attributes that have some degree of substructure. For example, it allows multivalued attributes such as phone number.
- When we create tables from E-R designs that contain the types of attributes as shown in figure, we eliminate this substructure.
- For composite attributes, we let each component be an attribute in its own right.
- For multivalued attributes,we create one tuple for each item in a multivalued set.



Atomic Domains and First Normal Form

- In the relational model, the idea is that, attributes do not have any substructure.
- A domain is atomic if elements of the domain are considered to be indivisible units.
- A relation schema R is in first normal form (1NF) if the domains of all attributes of R are atomic.
- For example, Integers are assumed to be atomic, so the set of integers is an atomic domain;
- however, the set of all sets of integers is a nonatomic domain.

Atomic Domains and First Normal Form

The important issue is not what the domain itself is, but rather how we use domain elements in our database.

Consider an organization that assigns employees identification numbers of the following form:

The first two letters specify the department and the remaining four digits are a unique number within the department for the employee. Examples of such numbers would be “CS001” and “EE1127”.

Such identification numbers can be divided into smaller units, and are therefore nonatomic.

If a relation schema had an attribute whose domain consists of identification numbers encoded as above, the schema would not be in first normal form.

Atomic Domains and First Normal Form

- The use of course identifiers such as “CS-101”, where “CS” indicates the Computer Science department, means that the domain of course identifiers is not atomic.
- However, the database application treats the domain as atomic, as long as it does not attempt to split the identifier and interpret parts of the identifier as a department abbreviation.
- The course schema stores the department name as a separate attribute, and the database application can use this attribute value to find the department of a course, instead of interpreting particular characters of the course identifier.
- Thus, our university schema can be considered to be in first normal form.

Atomic Domains and First Normal Form

- The use of set-valued attributes can lead to designs with redundant storage of data, which in turn can result in inconsistencies.
- Some types of nonatomic values can be useful but they should be used with care.
- For example, Composite-valued attributes are often useful, and set-valued attributes are also useful in many cases, which is why both are supported in the E-R model.

Decomposition Using Functional Dependencies

Keys and Functional Dependencies:

A database models, a set of entities and relationships in the real world. There are usually a variety of constraints (rules) on the data in the real world.

For example, some of the constraints that are expected to hold in a university database are:

1. Students and instructors are uniquely identified by their ID.

2. Each student and instructor has only one name.

3. Each instructor and student is (primarily) associated with only one department.

4. Each department has only one value for its budget, and only one associated building.

An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation; a legal instance of a database is one where all the relation instances are legal instances.

Keys and Functional Dependencies

Some of the most commonly used types of real-world constraints can be represented formally as keys (superkeys, candidate keys and primary keys), or as functional dependencies.

we defined the notion of a superkey as a set of one or more attributes that, taken collectively, allows us to identify uniquely a tuple in the relation.

We restate that definition here as follows:

Let r (R) be a relation schema. A subset K of R is a superkey of r (R) if, in any legal instance of r (R), for all pairs t1 and t2 of tuples in the instance of r if t1 ≠ t2, then t1[K] ≠ t2[K].

That is, no two tuples in any legal instance of relation r (R) may have the same value on attribute set K.

Keys and Functional Dependencies

Whereas a superkey is a set of attributes that uniquely identifies an entire tuple, a functional dependency allows us to express constraints that uniquely identify the values of certain attributes.

Consider a relation schema r (R), and let $\alpha \subseteq R$ and $\beta \subseteq R$.

- Given an instance of r (R), we say that the instance satisfies the functional dependency $\alpha \rightarrow \beta$ if for all pairs of tuples t_1 and t_2 in the instance such that

  $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t_1[\beta] = t_2[\beta]$.

- We say that the functional dependency $\alpha \rightarrow \beta$ holds on schema r (R) if, in every legal instance of r (R) it satisfies the functional dependency.

Using the functional-dependency notation, we say that K is a superkey of r (R) if the functional dependency $K \rightarrow R$ holds on r (R).

Keys and Functional Dependencies

In other words, K is a superkey if, for every legal instance of r (R), for every pair of tuples t_1 and t_2 from the instance, whenever $t_1[K] = t_2[K]$,

it is also the case that $t_1[R] = t_2[R]$ (that is, $t_1 = t_2$).

Functional dependencies allow us to express constraints that we cannot express with superkeys.

we considered the schema:

inst_dept (ID, name, salary, dept_name, building, budget)

in which the functional dependency *dept_name → budget* holds because for each department (identified by *dept_name*) there is a unique budget amount.

We denote the fact that the pair of attributes (*ID, dept_name*) forms a superkey for *inst_dept* by writing:

ID, dept_name→name, salary, building, budget

Keys and Functional Dependencies

We shall use functional dependencies in two ways:

1. To test instances of relations to see whether they satisfy a given set F of functional dependencies.

2. To specify constraints on the set of legal relations.We shall thus concern ourselves with only those relation instances that satisfy a given set of functional dependencies. If we wish to constrain ourselves to relations on schema r (R) that satisfy a set F of functional dependencies, we say that F holds on r (R).

Some functional dependencies are said to be trivial (little importance) because they are satisfied by all relations. For example, $A \to A$ is satisfied by all relations involving attribute A. Reading the definition of functional dependency literally,we see that,

for all tuples t1 and t2 such that $t1[A] = t2[A]$, it is the case that $t1[A] = t2[A]$.

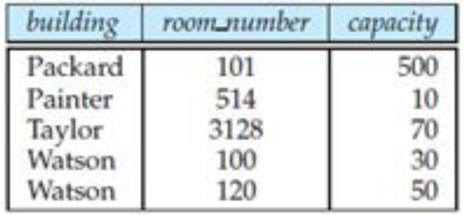
Similarly, $AB \to A$ is satisfied by all relations involving attribute A. In general, a functional dependency of the form $\alpha \to \beta$ is trivial if $\beta \subseteq \alpha$.

Keys and Functional Dependencies

It is important to realize that an instance of a relation may satisfy some functional dependencies that are not required to hold on the relation’s schema. In the instance of the classroom relation of Figure 8.5, we see that *room_number→capacity* is satisfied. However, in the real world, two classrooms in different buildings can have the same room number but with different room capacity.

Thus, it is possible, at some time, to have an instance of the classroom relation in which *room_number→capacity* is not satisfied. So, we would not include room number→capacity in the set of functional dependencies that hold on the schema for the *classroom* relation.

However, we would expect the functional dependency *building*, *room_number→capacity* to hold on the *classroom* schema.



building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure 8.5 An instance of the *classroom* relation.

Keys and Functional Dependencies

Given that a set of functional dependencies F holds on a relation r (R), it may be possible to infer that certain other functional dependencies must also hold on the relation.

For example, given a schema r (A, B,C), if functional dependencies A→ B and B → C, hold on r , we can infer the functional dependency A→ C must also hold on r. This is because, given any value of A there can be only one corresponding value for B, and for that value of B, there can only be one corresponding value for C.

We will use the notation F+ to denote the closure of the set F, that is, the set of all functional dependencies that can be inferred given the set F. Clearly F+ contains all of the functional dependencies in F.

Boyce–Codd Normal Form

- One of the more desirable normal forms that we can obtain is Boyce–Codd normal form (BCNF).

- It eliminates all redundancy that can be discovered based on functional dependencies, though, there may be other types of redundancy remaining.

Boyce–Codd Normal Form

A relation schema R is in BCNF with respect to a set F of functional dependencies if, for all functional dependencies in F+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency (that is, $\beta \subseteq \alpha$).
- α is a superkey for schema R.

A database design is in BCNF if each member of the set of relation schemas that constitutes the design is in BCNF.

Boyce–Codd Normal Form

The *instructor* schema is in BCNF. All of the nontrivial functional dependencies that hold, such as:

ID→name, dept name, salary

include *ID* on the left side of the arrow, and *ID* is a superkey (actually, in this case, the primary key) for *instructor*. (In other words, there is no nontrivial functional dependency with any combination of *name*, *dept_name*, and *salary*, without *ID*, on the side.) Thus, *instructor* is in BCNF.

Boyce–Codd Normal Form

Similarly, the *department* schema is in BCNF because all of the nontrivial functional dependencies that hold, such as:

dept_name→building, budget

include *dept_name* on the left side of the arrow, and *dept_name* is a superkey (and the primary key) for *department*. Thus, *department* is in BCNF.

Boyce–Codd Normal Form

To state a general rule for decomposing that are not in BCNF. Let R be a schema that is not in BCNF. Then there is at least one nontrivial functional dependency $\alpha \rightarrow \beta$ such that α is not a superkey for R.We replace R in our design with two schemas:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

In the case of *inst_dept* above, α = *dept_name*, β = {*building*, *budget*}, and *inst_dept* is replaced by

- $(\alpha \cup \beta)$ = (*dept_name, building,budget*)
- $(R - (\beta - \alpha))$ = (*ID, name, dept name, salary*)

In this example, it turns out that $\beta - \alpha = \beta$. We need to state the rule as we did so as to deal correctly with functional dependencies that have attributes that appear on both sides of the arrow.

Boyce–Codd Normal Form

Note that, When we decompose a schema that is not in BCNF, it may be that one or more of the resulting schemas are not in BCNF. In such cases, further decomposition is required, the eventual result of which is a set of BCNF schemas.

Difference between 1NF & 2NF

1NF

- The relation should not contain any composite or multi-valued attribute.
- The relation should contain only single valued attributes.
- A relation schema R is in first normal form (1NF) if the domains of all attributes of R are **atomic**.
- There should be a unique name to be specified for each attribute within the table.
- Example - student(rollno,sname,marks)

2NF

- The relations may contain composite keys.
- The relations should not contain any partial dependencies.
- The relations in 2NF handles the update anomalies.
- 2NF actually ensure the data dependencies.
- Example - course(stud_id, course_id, course_fee)

Difference between 3NF & BCNF

3NF

- The relation is said to be in 3NF when there is no transitive dependency.
- Lossless decomposition can be achieved by 3NF.

BCNF

- In BCNF for any relation A->B, A should be a super key of relation.
- Lossless decomposition is hard to achieve in BCNF.

Algorithms for Decomposition

BCNF Decomposition

- The definition of BCNF can be used directly to test if a relation is in BCNF. However, computation of F+ can be a tedious task.
- We first describe simplified tests for verifying if a relation is in BCNF. If a relation is not in BCNF, it can be decomposed to create relations that are in BCNF.
- Here we describe an algorithm to create a lossless decomposition of a relation, such that the decomposition is in BCNF.

Testing for BCNF

Testing of a relation schema R to see if it satisfies BCNF can be simplified in some cases:

- To check if a nontrivial dependency **$\alpha \rightarrow \beta$** causes a violation of BCNF, compute **α**+ (the attribute closure of), and verify that it includes all attributes of R; that is, it is a superkey of R.
- To check if a relation schema R is in BCNF, it suffices to check only the dependencies in the given set F for violation of BCNF, rather than check all dependencies in F+.

We can show that if none of the dependencies in F causes a violation of BCNF, then none of the dependencies in F+ will cause a violation of BCNF, either.

BCNF Decomposition

Unfortunately, the latter procedure does not work when a relation is decomposed. That is, it does not suffice to use F when we test a relation R_i , in a decomposition of R, for violation of BCNF.

For example, consider relation schema *R (A, B,C, D, E)*, with functional dependencies *F* containing $A \rightarrow B$ and $BC \rightarrow D$.

Suppose this were decomposed into *R1(A, B)* and *R2(A,C, D, E)*.

Now, neither of the dependencies in F contains only attributes from (A,C, D, E) so we might be misled into thinking R2 satisfies BCNF.

In fact, there is a dependency $AC \rightarrow D$ in *F+* (which can be inferred using the pseudotransitivity rule from the two dependencies in *F*) that shows that *R2* is not in BCNF.

Thus, we may need a dependency that is in *F+*, but is not in *F*, to show that a decomposed relation is not in BCNF.

BCNF Decomposition Algorithm

Till now we are able to state a general method to decompose a relation schema so as to satisfy BCNF.

Following shows an algorithm for this task

```
result := {R};
done := false;
compute F+;
while (not done) do
    if (there is a schema $R_i$ in result that is not in BCNF)
        then begin
            let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that holds
            on $R_i$ such that $\alpha \rightarrow$ Ri is not in F+, and $\alpha \cap \beta = \varnothing$;
            result := (result − $R_i$ ) ∪ ($R_i$ − $\beta$ ) ∪ ($\alpha$ , $\beta$ );
        End
    else done := true;
```

BCNF Decomposition Algorithm

If R is not in BCNF, we can decompose R into a collection of BCNF schemas R_1, R_2, . . . , R_n by the algorithm.

The algorithm uses dependencies that demonstrate violation of BCNF to perform the decomposition.

The decomposition that the algorithm generates is not only in BCNF, but is also a lossless decomposition.

BCNF Decomposition Algorithm

To see why our algorithm generates only lossless decompositions, we note that, when we replace a schema R_i with (Ri $-\boldsymbol{\beta}$) and ($\boldsymbol{\alpha}$,$\boldsymbol{\beta}$), the dependency $\boldsymbol{\alpha} \rightarrow \boldsymbol{\beta}$ holds, and (Ri $-\boldsymbol{\beta}$) ∩ ($\boldsymbol{\alpha}$,$\boldsymbol{\beta}$) = $\boldsymbol{\alpha}$

If we did not require $\boldsymbol{\alpha} \cap \boldsymbol{\beta} = \varnothing$, then those attributes in $\boldsymbol{\alpha} \cap \boldsymbol{\beta}$ would not appear in the schema (Ri $-\boldsymbol{\beta}$) and the dependency $\boldsymbol{\alpha} \rightarrow \boldsymbol{\beta}$ would no longer hold.

It is easy to see that our decomposition of *inst_dept* would result from applying the algorithm. The functional dependency *dept_name →building, budget* satisfies the $\boldsymbol{\alpha} \cap \boldsymbol{\beta} = \varnothing$ condition and would therefore be chosen to decompose the schema.

The BCNF decomposition algorithm takes time exponential in the size of the initial schema, since the algorithm for checking if a relation in the decomposition satisfies BCNF can take exponential time.

The Third Normal Form in DBMS

- A given relation is said to be in its third normal form when it's in 2NF but has no transitive partial dependency. Meaning, when no transitive dependency exists for the attributes that are non-prime, then the relation can be said to be in 3NF.
- A transitive dependency is a functional dependency in which A → C (A determines C) indirectly, because of A → B and B → C (where it is not the case that B → A). The third Normal Form ensures the reduction of data duplication.
- Lossless, dependency - preserving decomposition into 3NF is always possible.
- Every relation in BCNF is also in 3NF, but the reverse is not necessarily true. 3NF allows attributes to be part of a candidate key that is not the primary key; BCNF does not. This means that relations in 3NF are often in BCNF, but not always.

3NF Decomposition

An algorithm for finding a dependency-preserving, lossless decomposition into 3NF.

```
let $F_c$ be a canonical cover for F;
i := 0;
for each functional dependency $\alpha \rightarrow \beta$ in $F_c$
i := i + 1;
$R_i$ := $\alpha\beta$;
if none of the schemas $R_j$ , j = 1, 2, . . . , i contains a candidate key for R
then
i := i + 1;
$R_i$ := any candidate key for R;
/* Optionally, remove redundant relations */
repeat
if any schema $R_j$ is contained in another schema $R_k$
then
/* Delete $R_j$ */
$R_j$ := $R_i$ ;
i := i - 1;
until no more Rjs can be deleted
return ($R_1$, $R_2$, . . . , $R_i$ )
```

A canonical cover or irreducible a set of functional dependencies FD, is a simplified set of FD that has a similar closure as the original set FD.

https://chat.openai.com/share/ae693040-b632-409f-84c3-398ff6248da6

3NF Decomposition

The set of dependencies F_c used in the algorithm is a canonical cover for F.

Note that, the algorithm considers the set of schemas R_j , j = 1, 2, . . . , i;   initially i = 0, and in this case the set is empty.

Let us apply this algorithm to earlier example,

dept_advisor (s_ID, i_ID, dept_name)

is in 3NF even though it is not in BCNF.

The algorithm uses the following functional dependencies in F:

f1: i_ID→dept_name

f2: s_ID, dept_name→i_ID

3NF Decomposition

There are no extraneous attributes in any of the functional dependencies in F, so F_c contains f_1 and f_2.

The algorithm then generates as R_1 the schema, (i_ID, dept_name), and as R_2 the schema (s_ID, dept_name, i_ID).

The algorithm then finds that R_2 contains a candidate key, so no further relation schema is created.

3NF Decomposition

The resultant set of schemas can contain redundant schemas,with one schema R_k containing all the attributes of another schema R_j .

For example,

R_2 above contains all the attributes from R_1. The algorithm deletes all such schemas that are contained in another schema. Any dependencies that could be tested on an R_j that is deleted can also be tested on the corresponding relation R_k, and the decomposition is lossless even if R_j is deleted.

Comparison of BCNF and 3NF

Of the two normal forms for relational database schemas, 3NF and BCNF there are advantages to 3NF in that we know that it is always possible to obtain a 3NF design without sacrificing losslessness or dependency preservation.

Nevertheless, there are disadvantages to 3NF: We may have to use null values to represent some of the possible meaningful relationships among data items, and there is the problem of repetition of information.

Comparison of BCNF and 3NF

The goals of database design with functional dependencies are:

1. BCNF.
2. Losslessness.
3. Dependency preservation.

Since it is not always possible to satisfy all three, we may be forced to choose between BCNF and dependency preservation with 3NF.

Comparison of BCNF and 3NF

Given a BCNF decomposition that is not dependency preserving, we consider each dependency in a canonical cover F_c that is not preserved in the decomposition. For each such dependency $\alpha \rightarrow \beta$, we define a materialized view that computes a join of all relations in the decomposition, and projects the result on $\alpha\beta$. The functional dependency can be tested easily on the materialized view, using one of the constraints unique (α) or primary key (α).

On the negative side, there is a space and time overhead due to the materialized view, but on the positive side, the application programmer need not worry about writing code to keep redundant data consistent on updates; it is the job of the database system to maintain the materialized view, that is, keep it up to date when the database is updated.

Comparison of BCNF and 3NF

Thus, in case we are not able to get a dependency-preserving BCNF decomposition, it is generally preferable to opt for BCNF.