**Chapter 3**

**Relational Database Design**

**1 Informal Design Guidelines for Relational Databases**

**1.1 Semantics of the Relation Attributes**

**1.2 Redundant Information in Tuples and Update Anomalies**

**1.3 Null Values in Tuples**

**1.4 Spurious Tuples**

**2 Functional Dependencies (FDs)**

**2.1 Definition of FD**

**2.2 Inference Rules for FDs**

**2.3 Equivalence of Sets of FDs**

**3 Normal Forms Based on Primary Keys**

**3.1 Introduction to Normalization**

**3.2 First Normal Form**

**3.3 Second Normal Form**

**3.4 Third Normal Form**

**4 General Normal Form Definitions (For Multiple Keys)**

**5 BCNF (Boyce-Codd Normal Form)**

**Informal** **Design** **Guidelines** **for** **Relational** **Databases**

**GUIDELINE** **1:** **Informally, each tuple in a relation should represent one entity or relationship instance. (Apply to individual relations and their attributes).**

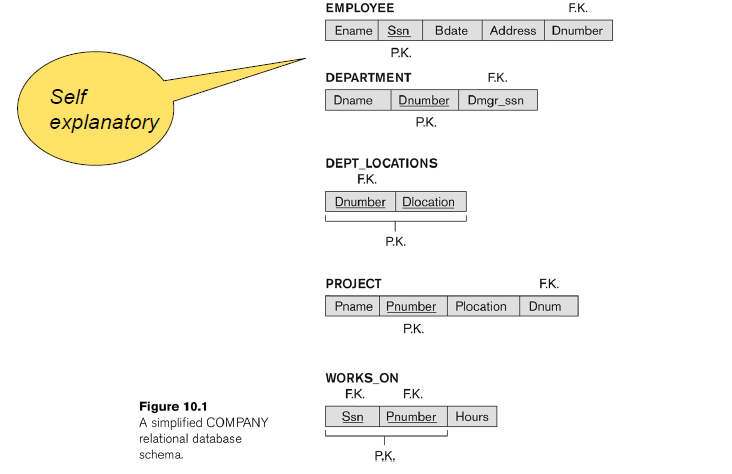
• Attributes of different entities (EMPLOYEEs, DEPARTMENTs, PROJECTs) should not be mixed in the same relation

• Only foreign keys should be used to refer to other entities

• Entity and relationship attributes should be kept apart as much as possible.

*Bottom Line:* Design a schema that can be explained easily relation by relation. The semantics of attributes should be easy to interpret.

**A Simplified Company Relational Schema**



**Redundant Information in Tuplesand Update Anomalies**

* **Big** **(and** **common)** **DB** **Problem:**

In a poorly designed‘ DB information is stored redundantly

* + **Consequences:**
  + Wastes storage
  + Causes problems with update anomalies

Insertion anomalies

Deletion anomalies

Modification anomalies

**EXAMPLE OF AN Update ANOMALY**

Consider the relation:

**EMP\_PROJ(Emp#,** **Proj#,** **Ename,** **Pname,** **No\_hours)**

**Update** **Anomaly:**

Changing the name of project number P1 from “Billing” to "Customer-Accounting"may cause this update to be made for all 100 employees working on project P1.

**EXAMPLE OF AN INSERT ANOMALY**

Consider the relation:

**EMP\_PROJ** **(Emp#,** **Proj#,** **Ename,** **Pname,** **No\_hours)**

**Insert** **Anomaly:**

Cannot insert a project unless an employee is assigned to the project

Conversely

Cannot insert an employee unless an he/she is assigned to a project.

**Figure of Two relation schemas suffering from update anomalies**

**Figure for Example State for and EMP\_PROJ**

**GUIDELINE** **2: Guideline to Redundant Information in and Update Anomalies**

* + Design a schema that does not suffer from the insertion, deletion and update anomalies.
  + If there are any anomalies present, then note them so that applications can be made to take them into account.

**GUIDELINE** **3: Null Values in Tuples**

* Relations should be designed such that their tuples will have as few NULL values as possible
* Attributes that are NULL frequently could be placed in separate relations (with the primary key)

**Reasons** **for** **nulls:**

* + Attribute not applicable or invalid
  + Attribute value unknown (may exist)
  + Value known to exist, but unavailable

**Spurious Tuples**

* Bad designs for a relational database may result in erroneous results for certain JOIN operations
* The "**lossless** **join**" property is used to guarantee meaningful results for join operations

**GUIDELINE** **4:**

* + The relations should be designed to satisfy the lossless join condition.
  + No spurious tuples should be generated by doing a natural-join of any relations.

There are two important properties of decompositions:

a) Non-additive or losslessness of the corresponding join

b) Preservation of the functional dependencies.

Note that:

* + Property (a) is extremely important and *cannot* be sacrificed.
  + Property (b) is less stringent and may be sacrificed.

**Example of Spurious Tuples**

**Example of Spurious Tuples**

**Functional Dependencies**

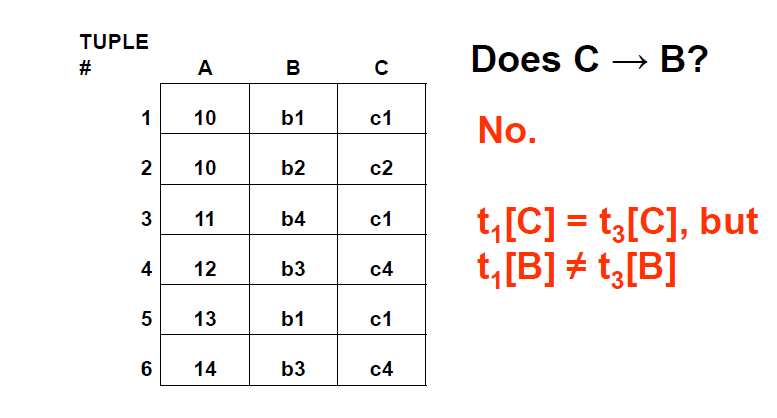
* Functional dependencies (FDs) are used to specify *formal measures* of the "goodness" of relational designs
* FDs and keys are used to define **normal forms** for relations
* FDs are **constraints** that are derived from the *meaning* and *interrelationships* of the data attributes
* FD is a constraint between two sets of attributes
* A set of attributes X *functionally determines* a set of attributes Y if the value of X determines a unique value for Y
* X 🡪 Y holds if whenever two tuples have the same value for X, they *must have* the same value for Y
* For any two tuples t1 and t2 in any relation instance r(R):

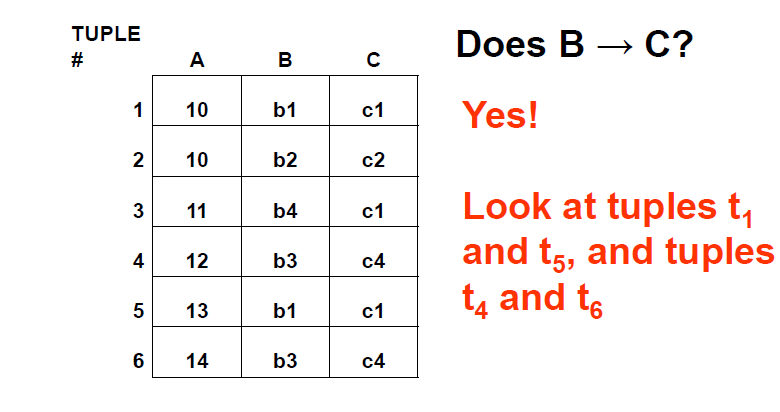
*If* t1[X]=t2[X],

*then* t1[Y]=t2[Y]

* X 🡪 Y in R specifies a *constraint* on all relation instances r(R)
* Written as X 🡪 Y; can be displayed graphically on a relation schema as in Figures. ( denoted by the arrow: ).
* FDs are derived from the real-world constraints on the attributes

**Example**





**Examples of FD constraints**

* social security number determines employee name SSN -> ENAME
* project number determines project name and location PNUMBER -> {PNAME, PLOCATION}
* employee ssn and project number determines the hours per week that the employee works on the project {SSN, PNUMBER} -> HOURS
* FDs must hold for all valid states of a relation, not just current state
* So define FDs carefully!

**How do we identify FDs?**

* Likely, some FDs will be obvious or identified in initial design of DB
* Vehicle(tagno, regstate, owner, make, model, year, gaseconomy, dealership, dealeraddr)

{tagno, regstate} → owner

{tagno, regstate} → {make, model, year}

{make, model, year} → gaseconomy,

dealership → dealeraddr etc…

* But the algorithms we use to test for other properties of good DB design often need to know **ALL** FDs!
* Some FDs may not be obvious, but can be deduced from other FDs
* Given a set of FDs F for a relation R, the set of all FDs for R is **F+**

(known as the ***closure*** of F)

**Inferring FDs**

Ex: (SSN, PNUMBER, HOURS, ENAME, PNAME, PLOCATION)

SSN → ENAME,

{SSN, PNUMBER} → HOURS, PNUMBER → PNAME, PNUMBER → PLOCATION

PNUMBER → PNAME,

so {PNUMBER, HOURS} → PNAME

PNUMBER → PNAME and PNUMBER → PLOCATION, so PNUMBER → {PNAME, PLOCATION}

**Inference Rules for FDs**

Given a set of FDs F, we can *infer* additional FDs that hold whenever the FDs in F hold

**Armstrong's inference rules:**

IR1. (**Reflexive**) If Y *subset-of* X, then X -> Y

IR2. (**Augmentation**) If X -> Y, then XZ -> YZ (Notation: XZ stands for X U Z)

IR3. (**Transitive**) If X -> Y and Y -> Z, then X -> Z

IR1, IR2, IR3 form a *sound* and *complete* set of inference rules

Some **additional inference rules** that are useful:

(**Decomposition**) If X -> YZ, then X -> Y and X -> Z

(**Union**) If X -> Y and X -> Z, then X -> YZ

(**Psuedo-transitivity**) If X -> Y and WY -> Z, then WX -> Z

* The last three inference rules, as well as any other inference rules, can be deduced from IR1, IR2, and IR3 (completeness property)

**Inference Rules for FDs**

* **Closure** of a set F of FDs is the set F+ of all FDs that can be inferred from F
* **Closure** of a set of attributes X with respect to F is the set X + of all attributes that are functionally determined by X
* X + can be calculated by repeatedly applying IR1, IR2, IR3 using the FDs in F

**Closure of X under F (X+)**

* X+ = set of all attributes dependent on X
* Algorithm
  1. start with X+ = X
  2. for each FD Y → Z in F do

if Y is a subset of X+ then X+ = X+ U Z

1. Continue this process until no more attributes can be added to X+

**Example**

**Given a relation Student and a set of functional dependencies F as follows, compute the closure for all LHS.**

Student(SID, dept, dept\_chair)

F = { SID → {dept, dept\_chair}, dept → dept\_chair,

{SID, dept} → dept\_chair }

{SID}+ = {SID, dept, dept\_chair}

{dept}+ = {dept, dept\_chair}

{SID, dept} + = {SID, dept, dept\_chair}

**If the closure of a LHS includes all attributes, then this LHS is a super key of the relation.**

**Candidate Keys**

* If X+ contains all attributes in a relation R, and if there does not exist Y in X

such that (X – Y)+ = all attributes in R, then X is a **candidate key** for R

* In previous example, {SID,dept}+ includes all attributes, SID+ also includes all attributes.
* SID is a candidate key

**Exercise**

R(A,B,C,D,G,H)

F = { A → B, B → C, CD → H, BC →G}

* What is the closure of AC?

**Minimal Sets of FDs**

* A set of FDs is **minimal** if it satisfies the following conditions:

1. Every dependency in F has a single attribute for its RHS.
2. We cannot remove any dependency from F and have a set of dependencies that is equivalent to F.
3. We cannot replace any dependency X -> A in F with a dependency Y -> A, where Y proper-subset-of X ( Y subset-of X) and still have a set of dependencies that is equivalent to F.
4. Every set of FDs has an equivalent minimal set
5. There can be several equivalent minimal sets
6. There is no simple algorithm for computing a minimal set of FDs that is equivalent to a set F of FDs

**Algorithm**

Given a set of FDs F, find its minimal cover

* Step1: Decompose each FD to get single attribute at RHS
* Step2: For each FD, remove redundant attribute from LHS
* Step3: Remove redundant FDs

**Example**

Given F= {B → AB, D → A, AB → D}

* Step 1: B → AB is decomposed into B → A, B → B

(B → B is trivial and is removed)

* Step 2: check if AB → D has redundant LHS. Can it be A → D or B → D?

Compute AB+, A+, B+ based on **F**

AB+ = ABD and B+ = ABD, so A is extraneous.

* So far, we have F= {B → A, D → A, B → D}
* So far, we have F= {B → A, D → A, B → D}
* Step 3: check if there is any redundant FDs. Is B → A redundant?

Compute B+ based on F- {B → A}

B+ = BDA, that means we can obtain

B → A from F- {B → A} so B → A is redundant.

Similarly, check the remaining FDs.

Final answer F’ = {D → A, B → D}

**Exercise**

Given a set of FDs F, find its **minimal cover**

* Step1: Decompose each FD to get single attribute at RHS
* Step2: For each FD, remove redundant attribute from LHS
* Step3: Remove redundant FDs

Question: what is the minimal cover of F?

R(A, B, C), F = {A → B, BC → A, AB → AC}

**Minimal Set (Cover) of FDs**

* There can be more than one minimal cover for a relation
* They won’t necessarily have the same number of FDs

**Normalization of Relations**

* **Normalization**: The process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations
* **Normal form**: Condition using keys and FDs of a relation to certify whether a relation schema is in a particular normal form
* **2NF, 3NF, BCNF**
  + based on keys and FDs of a relation schema
  + **4NF**
  + based on keys, multi-valued dependencies : MVDs; 5NF based on keys,
* Join dependencies : JDs (Chapter 11)
* Additional properties may be needed to ensure a good relational design (*lossless join, dependency preservation*; Chapter 11)

**Practical Use of Normal Forms**

* Normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties
* The practical utility of these normal forms is questionable when the constraints on which they are based are *hard to understand* or to *detect*
* The database designers *need not* normalize to the highest possible normal form
  + (usually up to 3NF, BCNF or 4NF)
* Denormalization:
  + The process of storing the join of higher normal form relations as a base relation—which is in a lower normal form

**Definitions of Keys and Attributes Participating in Keys**

* A **superkey** of a relation schema R = {A1, A2, ...., An} is a set of attributes S *subset-of* R with the property that no two tuples t1 and t2 in any legal relation state r of R will have t1[S] = t2[S]
* A **key** K is a superkey with the *additional property* that removal of any attribute from K will cause K not to be a super key any more.
* If a relation schema has more than one key, each is called a **candidate** key.
  + One of the candidate keys is *arbitrarily* designated to be the **primary key**, and the others are called secondary **keys**.
  + A **Prime attribute** must be a member of *some* candidate key
* A **Nonprime attribute** is not a prime attribute—that is, it is not a member of any candidate key.

**First Normal Form 1NF**

* A relation scheme R is in first normal form (1NF) if the values in *dom* (*A*) are atomic for every attribute A in R.
* **Disallows**
  + composite attributes
  + Set-valued attributes
  + **nested relations**; a cell of an *individual tuple* is a complex relation

Normalization nested relations into 1NF

**Second Normal Form**

* Uses the concepts of **FDs, primary key**
* **Definitions**
  + **Prime attribute:** An attribute that is member of the primary key K
  + **Left-Reduced or Full functional dependency:** a FD Y 🡪 Z where removal of any attribute from Y means the FD does not hold any more
* Examples:
  + {SSN, PNUMBER} 🡪 HOURS is a full FD since neither SSN " HOURS nor PNUMBER 🡪 HOURS hold
  + {SSN, PNUMBER} 🡪 ENAME is not a full FD (it is called a partial dependency ) since SSN 🡪ENAME also holds
* **A relation scheme R is in second normal form (2NF) with respect to a set of FDs F if it is in 1NF and every nonprime attribute is fully dependent on every key of R.**
* R can be decomposed into 2NF relations via the process of 2NF normalization
* **Example**

Let R=ABCD and F = { AB → C, B → D }. Here AB is a key. C and D are non-prime. C is fully dependent on the entire key AB, however D functionally depends on just *part* of the key (B →D). This is called a *partial dependency*

**Third Normal Form**

**Definition**:

Given a relation scheme R, a subset X of R, an attribute A in R, and a set of FDs F, A is **transitively dependent** upon X in R if there is a subset Y of R with:

X 🡪Y, Y🡪X and Y 🡪 A under F and AϵXY.

**Examples**:

Schema (ABCD) and F = {A 🡪B, B 🡪AC, C 🡪D }

D is transitively dependent on A(and B) via C, however C is not transitively dependent on A via B (B is prime).

* A relation schema R is in **third normal form (3NF)** if it is in 2NF *and* no non-prime attribute A in R is transitively dependent on the primary key
* R can be decomposed into 3NF relations via the process of 3NF normalization
* **NOTE**:
  + In X 🡪 Y and Y 🡪 Z, with X as the primary key, we consider this a problem only if Y is not a candidate key.
  + When Y is a candidate key, there is no problem with the transitive dependency .
  + E.g., Consider EMP (SSN, Emp#, Salary ).
* Here, SSN 🡪 Emp# 🡪 Salary and Emp# is a candidate key.

**Normalizing into 2NF and 3NF**

**Normal Forms Defined Informally**

* 1st normal form
  + All attributes depend on **the key**
* 2nd normal form
  + All attributes depend on **the whole key**
* 3rd normal form
  + All attributes depend on **nothing but the key**

**General Normal Form Definitions**

* The above definitions consider the primary key only
* The following more general definitions take into account relations with multiple candidate keys
* A relation schema R is in **second normal form (2NF)** if every non-prime attribute A in R is fully functionally dependent on *every* key of R
* **Example** Consider the schema
* SUPPLIER(sname, saddress, item, iname, price) and FDs
* F= { sname 🡪saddress, item 🡪iname, {sname, item} 🡪price }

1. *{sname, item}* is the primary key, all other attributes are non-prime.
2. Observe that *saddress* depends on part of the key (*sname*).
3. Likewise iname depends on part of the key (item)
4. Therefore SUPPLIER is not in 2NF

* Definition:
  + **Superkey** of relation schema R - a set of attributes S of R that contains a key of R
* A relation schema R is in **third normal form (3NF)** if whenever a FD X 🡪 A holds in R, then either:
  + X is a superkey of R, or
  + A is a prime attribute of R
* Consider an Example
* **Example** Key: { SID } and FD: SID→Building Building →Fee Building →Mgr

|  |  |  |  |
| --- | --- | --- | --- |
| **SID** | **Building** | **Fee** | **Manager** |
| 100 | Fenn | 300 | Mr. T |
| 300 | ABC | 400 | Ali |
| 200 | Holiday Inn | 400 | Tyson |

*Fee* (and *Manager*) transitively depend on *SID* via the non-prime attribute *Building*. Therefore the relation is not in 3NF.

* **Consider another Example**

|  |  |  |
| --- | --- | --- |
| name | address | beer |
| Sally  Sally | 123 Maple  123 Maple | Bud  Miller |

F= { name → address } Candidate key: (name, beer)

**Is this relation in 3NF?**

No: *name* is not a super key,

and *address* is not a part of any candidate key.

* **Consider another Example**

R(student, course, instructor) and F = {{student, course} → instructor, instructor → course}

Candidate key: (student, course)

It is in 3NF.

**Is 3NF Good Enough?**

* Still have **data redundancy**, consider following example
* **student,** **course, instructor**

(“John Doe”, “CS2300”, “McGeehan”)

(“Bob Jones”, “CS2300”, “McGeehan”)

This is Caused by the FD instructor → course where instructor is not a super key.

**BCNF (Boyce-Codd Normal Form)**

**Definition**

* A relation schema R is in **Boyce-Codd Normal Form (BCNF)** if whenever an **FD X**🡪**A** holds in R, then **X is a superkey** of R
* Consider an Example

Keys: { {Sid Major}, {Sid Fname} }

Sid, Major 🡪 Fname

Sid, Fname 🡪 Major

Fname 🡪Major

|  |  |  |
| --- | --- | --- |
| **SID** | **MAJOR** | **FNAME** |
| 100 | MATH | CAUCHY |
| 100 | PHYL | PLATO |
| 200 | MATH | CAUCHY |
| 300 | PHYS | NEWTON |
| 400 | PHYS | EINSTEIN |

* The relation is in 3NF but not in BCNF. Observe that Fname🡪Major is valid, but Fname is not a superkey.
* Problem: *Student 300 drops PHYS,* We lose information that says NEWTON is a PHYS advisor
* A solution: Decompose the relation into two (SID, FNAME), (FNAME, MAJOR)
* Each normal form is strictly stronger than the previous one
  + Every 2NF relation is in 1NF
  + Every 3NF relation is in 2NF
  + Every BCNF relation is in 3NF
  + There exist relations that are in 3NF but not in BCNF
* The goal is to have each relation in BCNF (or 3NF)

**Boyce-Codd Normal form**

A relation TEACH that is in 3NF but not in BCNF

**Achieving the BCNF by Decomposition**

* Three possible decompositions for relation TEACH
  + {student, instructor} and {student, course}
  + {course, instructor } and {course, student}
  + {instructor, course } and {instructor, student}
* All three decompositions will lose FD { student, course} 🡪 instructor
  + We have to settle for sacrificing the functional dependency preservation. But we cannot sacrifice the non-additivity property after decomposition.
* Out of the above three, only the 3rd decomposition will not generate spurious tuples after join.(and hence has the non-additivity property – to be discussed later) .