**Objectives**

At the end of this chapter the reader will be able to:

• Describe benefits of relational model

• Describe informal guidelines for relational schema

• Describe update, insertion and deletion anomalies

• Describe Functional Dependencies and its various forms

• Describe fundamentals of normalization

• Describe and distinguish the 1NF, 2NF, 3NF and BCNF normal forms.

1. **Introduction**

When designing a database, you have to make decisions regarding how best to take some system in the real world and model it in a database. This consists of deciding which tables to create, what columns they will contain, as well as the relationships between the tables. While it would be nice if this process was totally intuitive and obvious, or even better automated, this is simply not the case. A well-designed database takes time and effort to conceive, build and refine.

The benefits of a database that has been designed according to the relational model are numerous. Some of them are:

* Data entry, updates and deletions will be efficient.
* Data retrieval, summarization and reporting will also be efficient.
* Since the database follows a well-formulated model, it behaves predictably.
* Since much of the information is stored in the database rather than in the application, the database is somewhat self-documenting.
* Changes to the database schema are easy to make.

The objective of this chapter is to explain the basic principles behind relational database design and demonstrate how to apply these principles when designing a database.

1. **Informal Design Guidelines for Relational Schemas**

We discuss four *informal measures* of quality for relation schema design in this section:

* Semantics of the attributes
* Reducing the redundant values in tuples
* Reducing the null values in tuples
* Disallowing the possibility of generating spurious tuples

These measures are not always independent of one another, as we shall see.

**2.1 Semantics of the attributes**

The semantics, specifies how to interpret the attribute values stored in a tuple of the relation-in other words, how the attribute values in a tuple relate to one another. If the conceptual design is done carefully, followed by a systematic mapping into relations, most of the semantics will have been accounted for and the resulting design should have a clear meaning.

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 relation.

**2.2 Reducing the redundant values in tuples**

Storing the Same information redundantly, that is, in more than one place within a database, can lead to several problems:

**Redundant Storage:** Some information is stored repeatedly.

**Update Anomalies**: If one copy of such repeated data is updated, an inconsistency iscreated unless all copies are similarly updated.

**Insertion Anomalies**: It may not be possible to store certain information unless someother, unrelated, information is stored as well.

**Deletion Anomalies**: It may not be possible to delete certain information without losingsome other, unrelated, information as well.

Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations. If any anomalies are present, note them clearly and make sure that the programs that update the database will operate correctly.



**Figure 1 An Instance of Hourly\_Emps Relation**

**2.3 Reducing the null values in tuples**

It is worth considering whether the use of *null* values can address some of these problems. As we will see in the context of our example, they cannot provide a complete solution, but they can provide some help. In this chapter, we do not discuss the use of *null* values beyond this one example. Consider the example Hourly Elmps relation. Clearly, *null* values cannot help eliminate redundant storage or update anomalies. It appears thatthey can address insertion and deletion anomalies. For instance, to deal with the insertion anomaly example, we can insert an employee tuple with *null* values in the hourly wage field. However, *null* values cannot address all insertion anomalies. For example, we cannot record the hourly wage for a rating unless there is an employee with that rating, because we cannot store a null value in the *ssn* field, which is a primary key field. Similarly, to deal with the deletion anomaly examnple, we might consider storing a tuple with *null* values in all fields except *rating* and *hourly\_wages* if the last tuple with a given *rating* would otherwise be deleted. However, this solution does not work because itrequires the *8871,* value to be *null,* and primary key fields cannot be *null.* Thus, null values do not provide a general solution to the problems of redundancy, even though they can help in some cases.

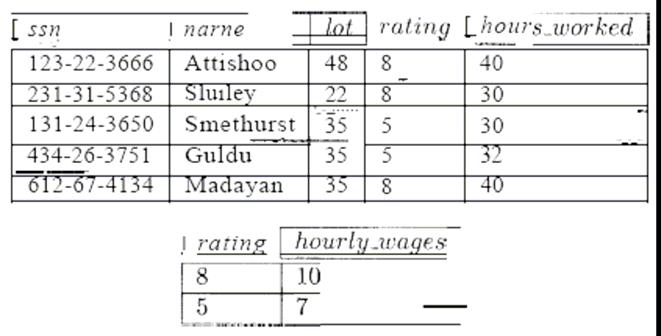
**Decompositions**

Intuitively, redundancy arises when a relational schema forces an association between attributes that is not natural. Functional dependencies can 'be used to identify such situations and suggest refinements to the schema. The essential idea is that many problems arising from redundancy can be addressed by replacing a relation 'with a collection of smaller relations. A. decomposition of a relation schema onsists of replacing the relation schema by two (or more) relation schema that each contain a subset of the attributes of *R* and together include all attributes in *R.* Intuitively, we want to store the information in any given instance of *R* by storing projections of the instance. This section examines the use of decompositions through several examples. we can decompose Hourly\_Emps into two relations:

Hourly\_Emps2(ssn,name,lot,rating\_hours\_*worked*)

Wages (*rating, hourly\_wages)*

The instances of these relations are shown in Figure 2 corresponding to instance shown in Figure 1



**Figure 2 Instance of Hourly\_Emps2 and Wages**

Note that we can easily record the hourly wage for any rating simply by adding a tuple to Wages, even if no employee with that rating appears in the current instance of Hourly\_Emps. Changing the wage associated with a rating involves updating a single Wages tuple. This is more efficient than updating several tuples (as in the original design), and it eliminates the potential for inconsistency.

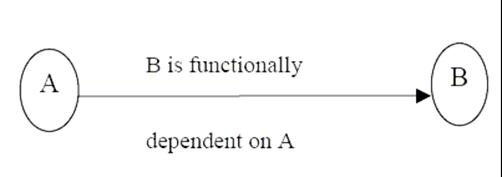
**2.4 Disallowing the possibility of generating spurious tuples**

Design relation schemas so that they can be joined with equality conditions on attributes that are either primary keys or foreign keys 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.

**3. Functional Dependencies**

For our discussion on functional dependencies assume that a relational schema has attributes (A, B, C... Z) and that the whole database is described by a single universal relation called R = (A, B, C, ..., Z). This assumption means that every attribute in the database has a unique name.

A functional dependency is a property of the semantics of the attributes in a relation. The semantics indicate how attributes relate to one another, and specify the functional dependencies between attributes. When a functional dependency is present, the dependency is specified as a constraint between the attributes.

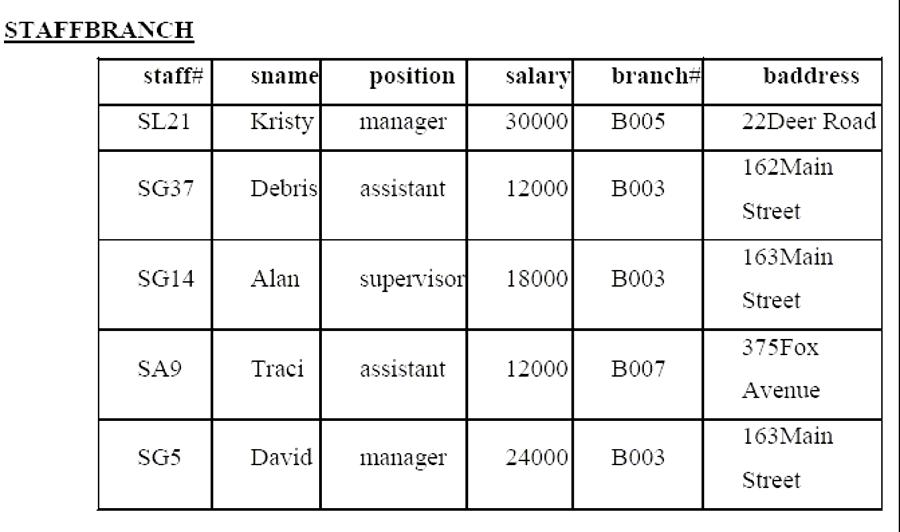
Consider a relation with attributes A and B, where attribute B is functionally dependent on attribute A. If we know the value of A and we examine the relation that holds this dependency, we will find only one value of B in all of the tuples that have a given value of A, at any moment in time. Note however, that for a given value of B there may be several different values of A.

In the above figure , A is the determinant of B and B is the consequent of A.

The determinant of a functional dependency is the attribute or group of attributes on the left-hand side of the arrow in the functional dependency. The consequent of a fd is the attribute or group of attributes on the right-hand side of the arrow.

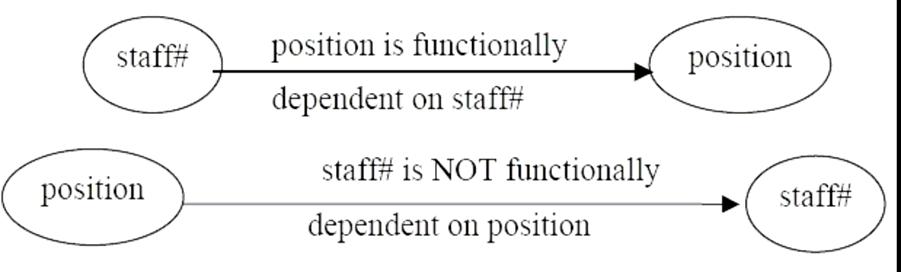
**Identifying Functional Dependencies**

Now let us consider the following Relational schema shown in figure



**Relational Schema**

The functional dependency staff# → position clearly holds on this relation instance. However, the reverse functional dependency position → staff# clearly does not hold. The relationship between staff# and position is 1:1 – for each staff member there is only one position. On the other hand, the relationship between position and staff# is 1:M – there are several staff numbers associated with a given position.



For the purposes of normalization we are interested in identifying functional dependencies between attributes of a relation that have a 1:1 relationship.

When identifying Fds between attributes in a relation it is important to distinguish clearly between the values held by an attribute at a given point in time and the set of all possible values that an attributes may hold at different times.

*In other words, a functional dependency is a property of a relational schema (its intension) and not a property of a particular instance of the schema (extension).*

The reason that we need to identify Fds that hold for all possible values for attributes of a relation is that these represent the types of integrity constraints that we need to identify. Such constraints indicate the limitations on the values that a relation can legitimately assume. In other words, they identify the legal instances which are possible.

Let’s identify the functional dependencies that hold using the relation schema STAFFBRANCH

In order to identify the time invariant Fds, we need to clearly understand the semantics of the various attributes in each of the relation schemas in question.

For example, if we know that a staff member’s position and the branch at which they are located determines their salary. There is no way of knowing this constraint unless you are familiar with the enterprise, but this is what the requirements analysis phase and the conceptual design phase are all about!

staff# → sname, position, salary, branch#, baddress

branch# → baddress

baddress → branch#

branch#, position → salary

baddress, position → salary

**3.1Trivial Functional Dependencies**

As well as identifying Fds which hold for all possible values of the attributes involved in the fd, we also want to ignore trivial functional dependencies. A functional dependency is trivial if, the consequent is a subset of the determinant. In other words, it is impossible for it not to be satisfied.

Example: Using the relation instances on page 6, the trivial dependencies include:

{ staff#, sname} → sname { staff#, sname} → staff#

Although trivial Fds are valid, they offer no additional information about integrity constraints for the relation. As far as normalization is concerned, trivial Fds are ignored.

**3.2 Inference Rules for Functional Dependencies**

We’ll denote as F, the set of functional dependencies that are specified on a relational schema R.

Typically, the schema designer specifies the Fds that are semantically obvious; usually however, numerous other Fds hold in all legal relation instances that satisfy the dependencies in F.

These additional Fds that hold are those Fds which can be inferred or deduced from the Fds in F.

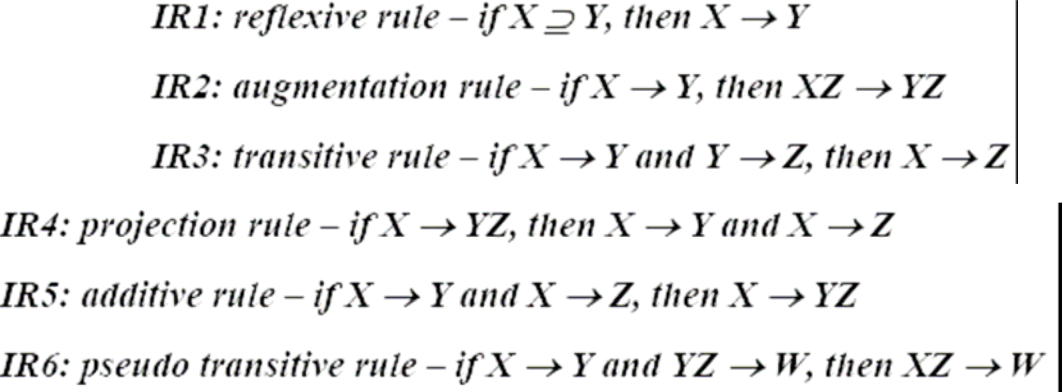
The set of all functional dependencies implied by a set of functional dependencies F is called the closure of F and is denoted F+.

The notation: F ⁭ X → Y denotes that the functional dependency X → Y is implied by the set of Fds F.

Formally, F+ ≡ {X → Y | F ⁭ X → Y}

A set of inference rules is required to infer the set of Fds in F+.

For example, if I tell you that Kristi is older than Debi and that Debi is older than Traci, you are able to infer that Kristi is older than Traci. How did you make this inference? Without thinking about it or maybe knowing about it, you utilized a transitivity rule to allow you to make this inference. The set of all Fds that are implied by a given set S of Fds is called the closure of S, written S+. Clearly we need an algorithm that will allow us to compute S+ from S. You know the first attack on this problem appeared in a paper by Armstrong which gives a set of inference rules. The following are the six well-known inference rules that apply to functional dependencies.



The first three of these rules (IR1-IR3) are known as Armstrong’s Axioms and constitute a necessary and sufficient set of inference rules for generating the closure of a set of functional dependencies. These rules can be stated in a variety of equivalent ways. Each of these rules can be directly proved from the definition of functional dependency. Moreover the rules are complete, in the sense that, given a set S of Fds, all Fds implied by S can be derived from S using the rules. The other rules are derived from these three rules.

Given R = (A,B,C,D,E,F,G,H, I, J) and

F={AB→E,AG→J,BE→I,E→G,GI→H} Does F ⁭ AB → GH?

**Proof**

* 1. AB → E, given in F
  2. AB → AB, reflexive rule IR1
  3. AB → B, projective rule IR4 from step 2
  4. AB → BE, additive rule IR5 from steps 1 and 3
  5. BE → I, given in F
  6. AB → I, transitive rule IR3 from steps 4 and 5
  7. E → G, given in F
  8. AB → G, transitive rule IR3 from steps 1 and 7
  9. AB → GI, additive rule IR5 from steps 6 and 8
  10. GI → H, given in F
  11. AB → H, transitive rule IR3 from steps 9 and 10

1. AB → GH, additive rule IR5 from steps 8 and 11 – proven

**Irreducible sets of dependencies**

Let S1 and S2 be two sets of Fds, if every FD implied by S1 is implied by S2- i.e.; if S1+ is a subset of S2+-we say that S2 is a cover for S1+(Cover here means equivalent set). What this means is that if the DBMS enforces the Fds in S2, then it will automatically be enforcing the Fds in S1.

Next if S2 is a cover for S1 and S1 is a cover for S2- i.e.; if S1+=S2+ -we say that S1 and S2 are equivalent, clearly, if s1 and S2 are equivalent, then if the DBMS enforces the Fds in S2 it will automatically be enforcing the Fds in S1, And vice versa.

Now we define a set of Fds to be irreducible( Usually called minimal in the literature) if and only if it satisfies the following three properties

1. The right hand side (the dependent) of every Fds in S involves just one attribute (that is, it is singleton set)
2. The left hand side (determinant) of every in S is irreducible in turn-meaning that no attribute can be discarded from the determinant without changing the closure S+(that is, with out converting S into some set not equivalent to S). We will say that such an Fd is **left irreducible**.
3. No Fd in S can be discarded from S without changing the closure S+(That is, without converting s into some set not equivalent to S)

Now we will work out the things in detail.

Relation R {A,B,C,D,E,F} satisfies the following Fds

AB→C

C → A

BC→D

ACD→B

BE→C

CE→FA

CF→VD

D→EF

Find an irreducible equivalent for this set of Fds?

Puzzled! The solution is simple. Let us find the solution for the above.

1. AB→C
2. C → A
3. BC→D
4. ACD→B
5. BE→C
6. CE→A
7. CE→F
8. CF→B
9. CF→D
10. D → E
11. D → F

Now:

* 2 implies 6, so we can drop 6
* 8 implies CF → BC (By augmentation), by which 3 implies CF → D (By Transitivity), so we can drop 10.
* 8 implies ACF → AB (By augmentation), and 11 implies ACD → ACF (By augmentation), and so ACD → AB (By Transitivity), and so ACD → B(By Decomposition), so we can drop 4

No further reductions are possible, and so we are left with the following irreducible set:

AB→C

C → A

BC→D

BE→C

CE→F

CF→B

D → E

D → F

Alternatively:

* 2 implies CD → ACD (By Composition), which with 4 implies CD → BE (By Transitivity), so we can replace 4 CD → B
* 2 implies 6, so we can drop 6(as before)

2 and 10 implies CF→ AD (By composition), which implies CF → ADC (By Augmentation), which with (the original) 4 implies CF → B(By Transitivity), So we can drop 8.

No further reductions are possible, and so we are left with following irreducible set:

AB→C

C → A

BC→D

CD→B

BE→C

CE→F

CF→D

D → E

D → F

Observe, therefore, that there are two distinct irreducible equivalence for the original set of Fds.

**4. Multivalued Dependencies**

The multivalued dependency relates to the problem when more than one multivalued attributes exist. Consider the following relation that represents an entity employee that has one mutlivalued attribute proj:

emp (e#, dept, salary, proj)

We have so far considered normalization based on functional dependencies; dependencies that apply only to single-valued facts. For example, **e#→dept** implies only one dept value for each value of e#. Not all information in a database is single-valued, for example, proj in an employee relation may be the list of all projects that the employee is currently working on. Although e# determines the list of all projects that an employee is working on **e#→proj**, is not a functional dependency.

So far we have dealt with multivalued facts about an entity by having a separate relation for that multivalue attribute and then inserting a tuple for each value of that fact. This resulted in composite keys since the multivalued fact must form part of the key. In none of our examples so far have we dealt with an entity having more than one multivalued attribute in one relation. We do so now.

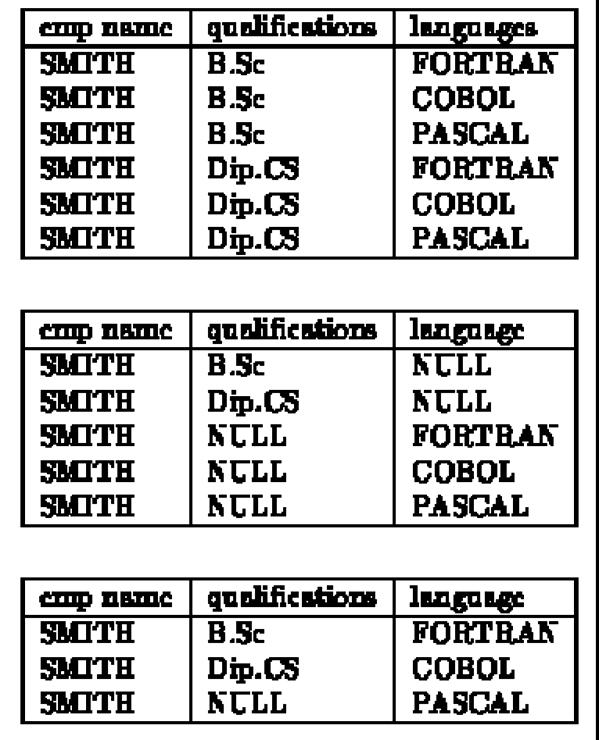
The fourth and fifth normal forms deal with multivalued dependencies. The 4 and 5 normal forms are discussed in the lecture that deals with normalization. We discuss the following example to illustrate the concept of multivalued dependency.

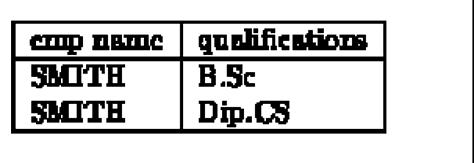
*programmer (emp\_name, qualifications, languages)*

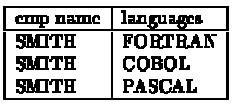
The above relation includes two multivalued attributes of entity programmer; *qualifications* and *languages*. There are no functional dependencies.

The attributes qualifications and languages are assumed independent of each other. If we were to consider qualifications and languages separate entities, we would have two relationships (one between *employees* and *qualifications* and the other between employees and programming languages). Both the above relationships are many-to-many i.e. one programmer could have several qualifications and may know several programming languages. Also one qualification may be obtained by several programmers and one programming language may be known to many programmers.

Suppose a programmer has several qualifications (B.Sc, Dip. Comp. Sc, etc) and is proficient in several programming languages; how should this information be represented? There are several possibilities.



Other variations are possible (we remind the reader that there is no relationship between qualifications and programming languages). All these variations have some disadvantages. If the information is repeated we face the same problems of repeated information and anomalies as we did when second or third normal form conditions are violated. If there is no repetition, there are still some difficulties with search, insertions and deletions. For example, the role of NULL values in the above relations is confusing. Also the candidate key in the above relations is (emp name, qualifications, language) and existential integrity requires that no NULLs be specified. These problems may be overcome by decomposing a relation like the one in above Figure as follows:



The basis of the above decomposition is the concept of multivalued dependency (MVD). Functional dependency **A→B** relates one value of A to one value of B while multivalued dependency **A→B** defines a relationship in which a set of values of attribute B are determined by a single value of A.

The concept of multivalued dependencies was developed to provide a basis for decomposition of relations like the one above. Therefore if a relation like *enrolment(sno,* *subject#)* has a relationship between *sno* and *subject#* in which sno uniquely determinesthe values of *subject#*, the dependence of *subject#* on sno is called a trivial MVD since the relation enrolment cannot be decomposed any further. More formally, a MVD **X→Y** is called *trivial* MVD if either Y is a subset of X or X and Y together form the relation R. The MVD is trivial since it results in no constraints being placed on the relation. Therefore a relation having non-trivial MVDs must have at least three attributes; two of them multivalued. Non-trivial MVDs result in the relation having some constraints on it since all possible combinations of the multivalue attributes are then required to be in the relation.

**Let us now define the concept of multivalued dependency**.

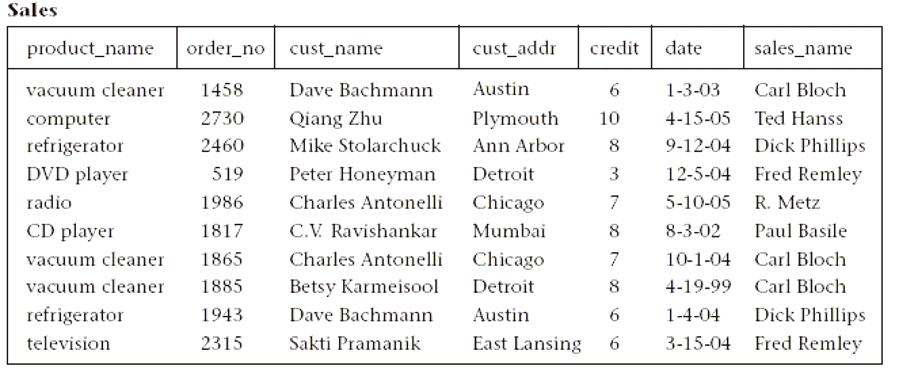
**The multivalued dependency X→Y is said to hold for a relation R(X, Y, Z) if for a given set of value (set of values if X is more than one attribute) for attributes X,**

**there is a set of (zero or more) associated values for the set of attributes Y and the Y values depend only on X values and have no dependence on the set of attributes Z.**

In the example above, if there was some dependence between the attributes *qualifications* and *language*, for example perhaps, the language was related to the qualifications (perhaps the qualification was a training certificate in a particular language), then the relation would not have MVD and could not be decomposed into two relations as above. The theory of multivalued dependencies in very similar to that for functional dependencies. Given D a set of MVDs, we may find **D+**, the closure of D using a set of axioms. We do not discuss the axioms here.

**5. Relational Database**

Relational database tables, whether they are derived from ER or UML models, sometimes suffer from some rather serious problems in terms of



**Single table database**

performance, integrity and maintainability. For example, when the entire database is defined as a single large table, it can result in a large amount of redundant data and lengthy searches for just a small number of target rows. It can also result in long and expensive updates, and deletions in particular can result in the elimination of useful data as an unwanted side effect.

Such a situation is shown in Single Table Database, where products, salespersons, customers, and orders are all stored in a single table called Sales. In this table, we see that certain product and customer information is stored redundantly, wasting storage space. Certain queries, such as “Which customers ordered vacuum cleaners last month?” would require a search of the entire table. Also, updates such as changing the address of the customer Dave Bachmann would require changing many rows. Finally, deleting an order by a valued customer such as Qiang Zhu (who bought an expensive computer), if that is his only outstanding order, deletes the only copy of his address and credit rating as a side effect. Such information may be difficult (or sometimes impossible) to recover. These problems also occur for situations in which the database has already been set up as a collection of many tables, but some of the tables are still too large.

If we had a method of breaking up such a large table into smaller tables so that these types of problems would be eliminated, the database would be much more efficient and reliable. Classes of relational database schemes or table definitions, called normal forms, are commonly used to accomplish this goal. The creation of a normal form database table is called **normalization**. Normalization is accomplished by analyzing the interdependencies among individual attributes associated with those tables and taking projections (subsets of columns) of larger tables to form smaller ones.

Normalization is a formal process for determining which fields belong in which tables in a relational database. Normalization follows a set of rules worked out at the time relational databases were born. A normalized relational database provides several benefits:

* Elimination of redundant data storage.
* Close modeling of real world entities, processes, and their relationships.
* Structuring of data so that the model is flexible.
* Normalization ensures that you get the benefits relational databases offer. Time spent learning about normalization will begin paying for itself immediately.

Let us first review the basic normal forms, which have been well established in the relational database.

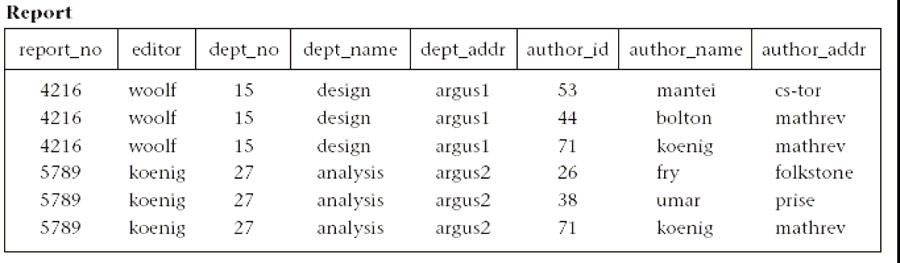
**6. First Normal Form**

***Definition.*** A table is in*first normal form (1NF)*if and only if all columns contain onlyatomic values, that is, each column can have only one value for each row in the table. Relational database tables, such as the **Sales** table illustrated in Report Tables, have only atomic values for each row and for each column. Such tables are considered to be in first normal form, the most basic level of normalized tables.

To better understand the definition for 1NF, it helps to know the difference between a domain, an attribute, and a column. A *domain* is the set of all possible values for a particular type of attribute, but may be used for more than one attribute. For example, the domain of people’s names is the underlying set of all possible names that could be used for either customer-name or salesperson-name in the database table in Single Table Database. Each column in a relational table represents a single attribute, but in some cases more than one column may refer to different attributes from the same domain. When this occurs, the table is still in 1NF because the values in the table are still atomic. In fact, standard SQL assumes only atomic values and a relational table is by default in 1NF.

**6.1 Superkeys, Candidate Keys, and Primary Keys**

A table in 1NF often suffers from data duplication, update performance, and update integrity problems, as noted above. To understand these issues better, however, we must define the concept of a key in the context of normalized tables. A *superkey* is a set of one or more attributes, which, when taken collectively, allows us to identify uniquely an entity or table. Any subset of the attributes of a superkey that is also a superkey, and not reducible to another superkey, is called a *candidate key*. A *primary* key is selected arbitrarily from the set of candidate keys to be used in an index for that table.



**Report table**

As an example, in above Figure a composite of all the attributes of the table forms a superkey because duplicate rows are not allowed in the relational model. Thus, a trivial superkey is formed from the composite of all attributes in a table. Assuming that each department address (dept\_addr) in this table is single valued, we can conclude that the composite of all attributes except dept\_addr is also a superkey. Looking at smaller and smaller composites of attributes and making realistic assumptions about which attributes are single valued, we find that the composite (report\_no, author\_id) uniquely determines all the other attributes in the table and is therefore a superkey. However, neither report\_no nor author\_id alone can determine a row uniquely, and the composite of these two attributes cannot be reduced and still be a superkey. Thus, the composite (report\_no, author\_id) becomes a candidate key. Since it is the only candidate key in this table, it also becomes the primary key.

A table can have more than one candidate key. If, for example, in Report Table, we had an additional column for author\_ssn, and the composite of report\_no and author\_ssn uniquely determine all the other attributes of the table, then both (report\_no, author\_id) and (report\_no, author\_ssn) would be candidate keys. The primary key would then be an arbitrary choice between these two candidate keys.

**7. Second Normal Form**

To explain the concept of second normal form (2NF) and higher, we introduce the concept of functional dependence. The property of one or more attributes that uniquely determine the value of one or more other attributes is called *functional dependence*. Given a table (R), a set of attributes (B) is functionally dependent on another set of attributes (A) if, at each instant of time, each A value is associated with only one B value. Such a functional dependence is denoted by A -> B. In the preceding example from REPORT TABLE, let us assume we are given the following functional dependencies for the table **report**:

***report:*** report\_no -> editor, dept\_no

dept\_no -> dept\_name, dept\_addr

author\_id -> author\_name, author\_addr

***Definition.*** A table is in second normal form (2NF) if and only if it is in 1NF and everynonkey attribute is fully dependent on the primary key. An attribute is fully dependent on the primary key if it is on the right side of an FD for which the left side is either the primary key itself or something that can be derived from the primary key using the transitivity of FDs.

An example of a transitive FD in **report** is the following:

report\_no -> dept\_no

dept\_no -> dept\_name

Therefore we can derive the FD (report\_no -> dept\_name), since dept\_name is transitively dependent on report\_no.

Continuing our example, the composite key in Report Table, (report\_no, author\_id), is the only candidate key and is therefore the primary key. However, there exists one FD (dept\_no -> dept\_name, dept\_addr) that has no component of the primary key on the left side, and two FDs (report\_no -> editor, dept\_no and author\_id -> author\_name, author\_addr) that contain one component of the primary key on the left side, but not both components. As such, **report** does not satisfy the condition for 2NF for any of the FDs.

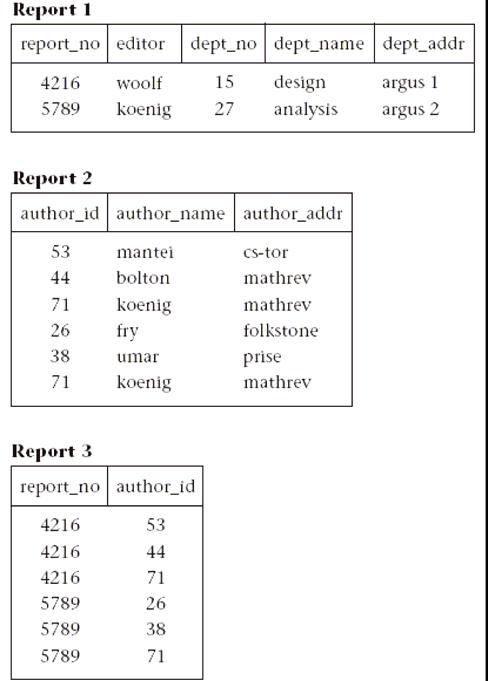
Consider the disadvantages of 1NF in table **report**. Report\_no, editor, and dept\_no are duplicated for each author of the report. Therefore, if the editor of the report changes, for example, several rows must be updated. This is known as the *update anomaly*, and it represents a potential degradation of performance due to the redundant updating. If a new editor is to be added to the table, it can only be done if the new editor is editing a report: both the report number and editor number must be known to add a row to the table, because you cannot have a primary key with a null value in most relational databases. This is known as the *insert anomaly*. Finally, if a report is withdrawn, all rows associated with that report must be deleted. This has the side effect of deleting the information that associates an author\_id with author\_name and author\_addr. Deletion side effects of this nature are known as *delete anomalies*. They represent a potential loss of integrity, because the only way the data can be restored is to find the data somewhere outside the database and insert it back into the database. All three of these anomalies represent problems to database designers, but the delete anomaly is by far the most serious because you might lose data that cannot be recovered. These disadvantages can be overcome by transforming the 1NF table into two or more 2NF tables by using the projection operator on the subset of the attributes of the 1NF table. In this example we project **report** over report\_no, editor, dept\_no, dept\_name, and dept\_addr to form **report1**; and project **report** over author\_id, author\_name, and author\_addr to form **report2**; and finallyproject **report** over report\_no and author\_id to form **report3**. The projection of **report** into three smaller tables has preserved the FDs and the association between report\_no and author\_no that was important in the original table. Data for the three tables is shown in 2NF Tables. The FDs for these 2NF tables are: ***report1:*** report\_no -> editor, dept\_no

dept\_no -> dept\_name, dept\_addr

***report2:*** author\_id -> author\_name, author\_addr

***report3:*** report\_no, author\_id is a candidate key (no FDs)

We now have three tables that satisfy the conditions for 2NF, and we have eliminated the worst problems of 1NF, especially integrity (the delete anomaly). First, editor, dept\_no, dept\_name, and dept\_addr are no longer duplicated for each author of a report. Second, an editor change results in only an update to one row for **report1**. And third, the most important, the deletion of the report does not have the side effect of deleting the author information.

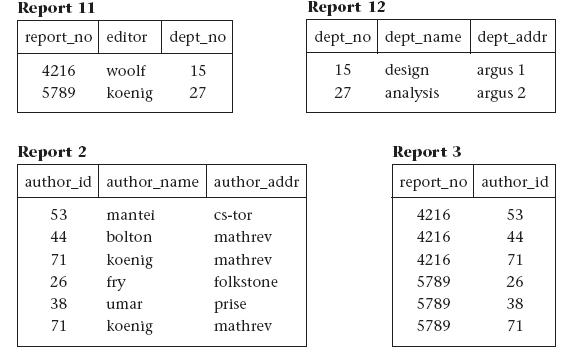


**2NF tables**

Not all performance degradation is eliminated, however; report\_no is still duplicated for each author, and deletion of a report requires updates to two tables (**report1** and **report3**) instead of one. However, these are minor problems compared to those in the 1NF table **report**. Note that these three report tables in 2NF could have been generated directlyfrom an ER (or UML) diagram that equivalently modeled this situation with entities Author and Report and a many-to-many relationship between them.

**8. Third Normal Form**

The 2NF tables we established in the previous section represent a significant improvement over 1NF tables. However, they still suffer from



**3NF tables**

the same types of anomalies as the 1NF tables although for different reasons associated with transitive dependencies. If a transitive (functional) dependency exists in a table, it means that two separate facts are represented in that table, one fact for each functional dependency involving a different left side. For example, if we delete a report from the database, which involves deleting the appropriate rows from **report1** and **report3** (see 2NF TABLES), we have the side effect of deleting the association between dept\_no, dept\_name, and dept\_addr as well. If we could project table **report1** over report\_no, editor, and dept\_no to form table **report11**, and project **report1** over dept\_no, dept\_name, and dept\_addr to form table **report12**, we could eliminate this problem. Example tables for **report11** and **report12** are shown in 3NF Tables.

***Definition.*** A table is in*third normal form (3NF)*if and only if for every nontrivialfunctional dependency X->A, where X and A are either simple or composite attributes, one of two conditions must hold. Either attribute X is a superkey, or attribute A is a member of a candidate key. If attribute A is a member of a candidate key, A is called a prime attribute. Note: a trivial FD is of the form YZ->Z. In the preceding example, after projecting **report1** into **report11** and **report12** to eliminate the transitive dependency report\_no -> dept\_no -> dept\_name, dept\_addr, we have the following 3NF tables and their functional dependencies (and example data in 3NF Tables):

***report11:*** report\_no -> editor, dept\_no

***report12:*** dept\_no -> dept\_name, dept\_addr

***report2:*** author\_id -> author\_name, author\_addr

***report3:*** report\_no, author\_id is a candidate key (no FDs)

**9. Boyce-Codd Normal Form**

3NF, which eliminates most of the anomalies known in databases today, is the most common standard for normalization in commercial databases and CASE tools. The few remaining anomalies can be eliminated by the Boyce-Codd normal form (BCNF). BCNF is considered to be a strong variation of 3NF.

***Definition.*** A table**R**is in*Boyce-Codd normal form (BCNF)*if for every nontrivial FDX->A, X is a superkey.

BCNF is a stronger form of normalization than 3NF because it eliminates the second condition for 3NF, which allowed the right side of the FD to be a prime attribute. Thus, every left side of an FD in a table must be a superkey. Every table that is BCNF is also 3NF, 2NF, and 1NF, by the previous definitions.

The following example shows a 3NF table that is not BCNF. Such tables have delete anomalies similar to those in the lower normal forms.

***Assertion 1.*** For a given team, each employee is directed by only one leader. A team may

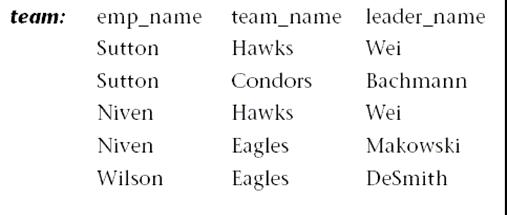
be directed by more than one leader.

emp\_name, team\_name -> leader\_name

***Assertion 2.*** Each leader directs only one team.

leader\_name -> team\_name

This table is 3NF with a composite candidate key emp\_id, team\_id:



The **team** table has the following delete anomaly: if Sutton drops out of the Condors team, then we have no record of Bachmann leading the Condors team. the natural join of those subset tables must result in the original table without any extra unwanted rows.

**Assignment Questions**

1. What are the various guidelines that need to be taken care of while designing a relational schema?
2. Describe update, insert and delete anomalies with the help of examples.
3. Define functional dependencies?
4. Explain the inference rules?
5. Explain the concept of multi valued dependency?
6. What does the term unnormalized relation refer to? How did the normal forms develop historically?
7. Write down all the rules for normalization and explain with example.
8. Define first, second, and third normal forms when only primary keys are considered. How do the general definitions of 2NF and 3NF, which consider all keys of a relation, differ from those that consider only primary keys?