Chapter 5: FUNCTIONAL DEPENDENCIES AND NORMALIZATION FOR RELATIONAL DATABASES

INFORMAL DESIGN GUIDELINES FOR RELATION SCHEMAS

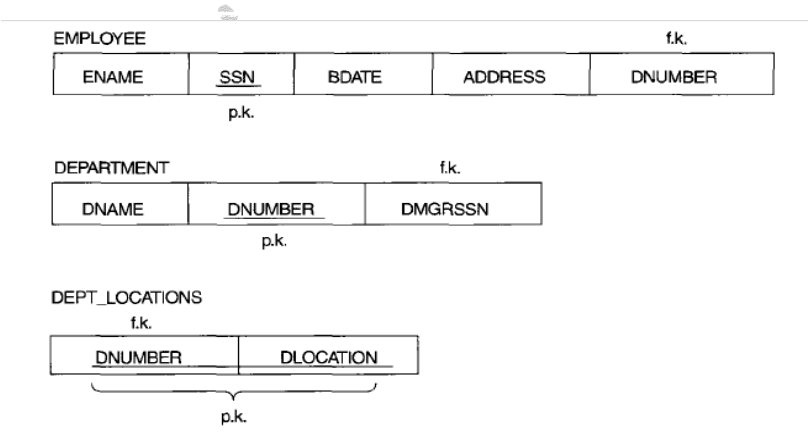
We discuss four *informal measures* of quality for relation schema design in this section:

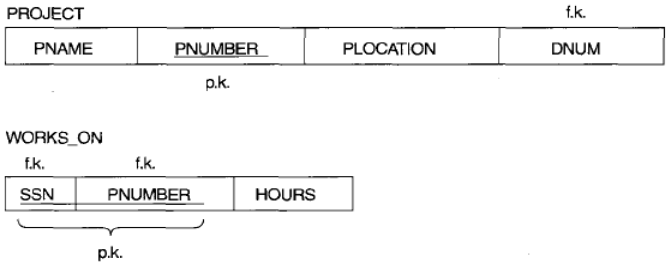
- Semantics of the attributes
- Reducing the redundant values in tuples
- Reducing the null values in tuples
- Disallowing the possibility of generating spurious tuples

Semantics of the Relation Attributes

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. In general, the easier it is to explain the semantics of the relation, the better the relation schema design will be.

To illustrate this, consider a simplified version of the COMPANY relational database schema. The meaning of the EMPLOYEE relation schema is quite simple: Each tuple represents an employee, with values for the employee's name (ENAME), social security number (SSN), birth date (BDATE), and address (ADDRESS), and the number of the department that the employee works for (DNUMBER). The DNUMBER attribute is a foreign key that represents an *implicit relationship* between EMPLOYEE and DEPARTMENT. The semantics of the DEPARTMENT and PROJECT schemas are also straightforward: Each DEPARTMENT tuple represents a department entity, and each PROJECT tuple represents a project entity. The attribute DMGRSSN of DEPARTMENT relates a department to the employee who is its manager, while DNUM of PROJECT relates a project to its controlling department; both are foreign key attributes. The ease with which the meaning of a relation's attributes can be explained is an *informal measure* of how well the relation is designed.





GUIDELINE 1. Design a relation schema so that it is easy to explain its meaning. Do not combine attributes from multiple entity types and relationship types into a single relation. Intuitively, if a relation schema corresponds to one entity type or one relationship type, it is straightforward to explain its meaning. Otherwise, if the relation corresponds to a mixture of multiple entities and relationships, semantic ambiguities will result and the relation cannot be easily explained.

Redundant Information in Tuples and Update Anomalies

One goal of schema design is to minimize the storage space used by the base relations. Grouping attributes into relation schemas has a significant effect on storage space. For example, compare the space used by the two base relations EMPLOYEE and DEPARTMENT in Figure A with that for an EMP_DEPT base relation in Figure B which is the result of applying the NATURAL JOIN operation to EMPLOYEE and DEPARTMENT.

EMPLOYEE

ENAME	SSN	BDATE	ADDRESS	DNUMBER
Smith,John B.	123456789	1965-01-09	731 Fondren,Houston,TX	5
Wong,Franklin T.	333445555	1955-12-08	638 Voss,Houston,TX	5
Zelaya,Alicia J.	999887777	1968-07-19	3321 Castle,Spring,TX	4
Wallace,Jennifer S.	987654321	1941-06-20	291 Berry,Bellaire,TX	4
Narayan,Ramesh K.	666884444	1962-09-15	975 Fire Oak,Humble,TX	5
English,Joyce A.	453453453	1972-07-31	5631 Rice,Houston,TX	5
Jabbar,Ahmad V.	987987987	1969-03-29	980 Dallas,Houston,TX	4
Borg,James E.	888665555	1937-11-10	450 Stone,Houston,TX	1

DEPARTMENT

DNAME	DNUMBER	DMGRSSN
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

Figure A: EMPLOYEE and DEPARTMENT Relations

EMP_DEPT

redundancy

ENAME	SSN	BDATE	ADDRESS	DNUMBER	DNAME	DMGRSSN
Smith,John B.	123456789	1965-01-09	731 Fondren,Houston,TX	5	Research	333445555
Wong,Franklin T.	333445555	1955-12-08	638 Voss,Houston,TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle,Spring,TX	4	Administration	987654321
Wallace,Jennifer S.	987654321	1941-06-20	291 Berry,Bellaire,TX	4	Administration	987654321
Narayan,Ramesh K.	666884444	1962-09-15	975 FireOak,Humble,TX	5	Research	333445555
English,Joyce A.	453453453	1972-07-31	5631 Rice,Houston,TX	5	Research	333445555
Jabbar,Ahmad V.	987987987	1969-03-29	980 Dallas,Houston,TX	4	Administration	987654321
Borg,James E.	888665555	1937-11-10	450 Stone,Houston,TX	1	Headquarters	888665555

Figure B: Natural join on EMPLOYEE & DEPARTMENT

In EMP_DEPT, the attribute values pertaining to a particular department (DNUMBER, DNAME, DMGRSSN) are repeated for *every employee who works for that department.* In contrast, each department's information appears only once in the DEPARTMENT relation in Figure A. Only the department number (DNUMBER) is repeated in the EMPLOYEE relation for each employee who works in that department.

Another serious problem with using the relations in Figure B as base relations is the problem of update anomalies. These can be classified into insertion anomalies, deletion anomalies, and modification anomalies.

Insertion Anomalies: Insertion anomalies can be differentiated into two types, based on the EMP_DEPT relation:

- To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for the department that the employee works for, or nulls (if the employee does not work for a department as yet).
- It is difficult to insert a new department that has no employees as yet in the EMP_DEPT relation. The only way to do this is to place null values in the attributes for employee. This causes a problem because SSN is the primary key of EMP_DEPT, and each tuple is supposed to represent an employee entity-not a department entity.

Deletion Anomalies: If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost from the database. This problem does not occur in the database of Figure A because DEPARTMENT tuples are stored separately.

Modification Anomalies: In EMP_DEPT, if we change the value of one of the attributes of a particular department-say, the manager of department 5-we must update the tuples of all employees who work in that department; otherwise, the database will become inconsistent. If we fail to update some tuples, the same department will be shown to have two different values for manager in different employee tuples, which would be wrong.

GUIDELINE 2. Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations. If any anomalies are present, note them clearly and make sure that the programs that update the database will operate correctly.

Null Values in Tuples

If many of the attributes do not apply to all tuples in the relation, we end up with many nulls in those tuples. This can waste space at the storage level and may also lead to problems with understanding the meaning of the attributes and with specifying JOIN operations at the logical level. Another problem with nulls is how to account for them when aggregate operations such as COUNT or SUM are applied. Moreover, nulls can have multiple interpretations, such as the following:

• The attribute *does not apply* to this tuple.

• The attribute value for this tuple is *unknown.*

• The value is *known but absent;* that is, it has not been recorded yet.

Having the same representation for all nulls compromises the different meanings they may have. Therefore, we may state another guideline.

GUIDELINE 3: As far as possible, avoid placing attributes in a base relation whose values may frequently be null. If nulls are unavoidable, make sure that they apply in exceptional cases only and do not apply to a majority of tuples in the relation.

Generation of Spurious Tuples

Consider the two relation schemas EMP_LOCS and EMP_PROJ1 in Figure 2A, which can be used instead of the single EMP_PROJ relation of Figure 2B. A tuple in EMP_LOCS means that the employee whose name is ENAME works on *some project* whose location is PLOCATION.

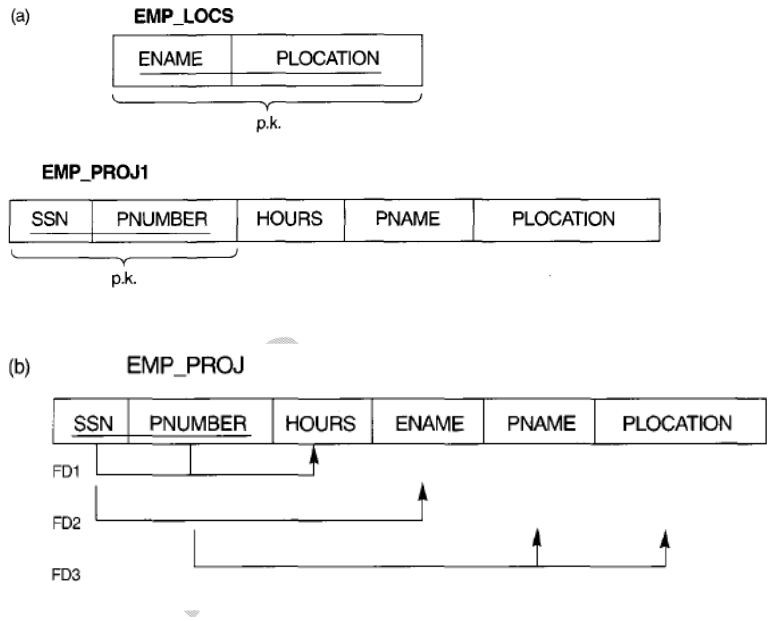


Figure 2

A tuple in EMP_PROJ1 means that the employee whose social security number is SSN works HOURS per week on the project whose name, number, and location are PNAME, PNUMBER, and PLOCATION. Suppose that we used EMP_PROJ1 and EMP_LOCS as the base relations instead of EMP_PROJ. This produces a particularly bad schema design, because we cannot recover the information that was originally in EMP_PROJ from EMP_PROJ1 and EMP_LOCS. If we attempt a NATURALJOIN operation on EMP_PROJ1 and EMP_LOCS, the result produces many more tuples than the original set of tuples in EMP_PROJ.

Additional tuples that were not in EMP_PROJ are called spurious tuples because they represent spurious or *wrong* information that is not valid.
Decomposing EMP_PROJ into EMP_LOCS and EMP_PROJ1 is undesirable because, when we JOIN them back using NATURAL JOIN, we do not get the correct original information. This is because in this case PLOCATION is the attribute that relates EMP_LOCS and EMP_PROJ1, and PLOCATION is neither a primary key nor a foreign key in either EMP_LOCS or EMP_PROJ1.We can now informally state another design guideline.

GUIDELINE 4: 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.

Summary and Discussion of Design Guidelines

We informally discussed situations that lead to problematic relation schemas, and we proposed informal guidelines for a good relational design. The problems we pointed out, which can be detected without additional tools of analysis, are as follows:

- Anomalies that cause redundant work to be done during insertion into and modification of a relation, and that may cause accidental loss of information during a deletion from a relation
- Waste of storage space due to nulls and the difficulty of performing aggregation operations and joins due to null values
- Generation of invalid and spurious data during joins on improperly related base relations

FUNCTIONAL DEPENDENCIES

A **functional dependency** (FD) is a constraint between two sets of attributes in a relation from a database.

Given a relation *R*, a set of attributes *X* in *R* is said to **functionally determine** another attribute *Y*, also in *R*, (written $X \rightarrow Y$) if and only if each *X* value is associated with precisely one *Y* value. Customarily we call *X* the *determinant set* and *Y* the *dependent attribute*. Thus, given a tuple and the values of the attributes in *X*, one can determine the corresponding value of the *Y* attribute. For the purposes of simplicity, given that *X* and *Y* are sets of attributes in *R*, $X \rightarrow Y$ denotes that *X* functionally determines each of the members of *Y* - in this case *Y* is known as the *dependent set*. Thus, a candidate key is a minimal set of attributes that functionally determine all of the attributes in a relation.

A functional dependency FD: $X \rightarrow Y$ is called **trivial** if Y is a subset of X.

The determination of functional dependencies is an important part of designing databases in the relational model, and in database normalization and denormalization. The functional dependencies, along with the attribute domains, are selected so as to generate constraints that would exclude as much data inappropriate to the user domain from the system as possible.

For example, suppose one is designing a system to track vehicles and the capacity of their engines. Each vehicle has a unique vehicle identification number (VIN). One would write **VIN** → **EngineCapacity** because it would be inappropriate for a vehicle's engine to have more than one capacity. (Assuming, in this case, that vehicles only have one engine.) However, *EngineCapacity* → *VIN*, is incorrect because there could be many vehicles with the same engine capacity.

This functional dependency may suggest that the attribute EngineCapacity be placed in a relation with candidate key VIN. However, that may not always be appropriate. For example, if that functional dependency occurs as a result of the transitive functional dependencies VIN → VehicleModel and VehicleModel → EngineCapacity then that would not result in a normalized relation.

Irreducible function depending set

Functional depending set S is irreducible if the set has three following properties:

- Each right set of a functional dependency of S contains only one attribute.
- Each left set of a functional dependency of S is irreducible. It means that reducing any one attribute from left set will change the content of S (S will lose some information).
- Reducing any functional dependency will change the content of S.

Sets of Functional Dependencies(FD) with these properties are also called *canonical* or *minimal*.

Properties of functional dependencies

Given that *X*, *Y*, and *Z* are sets of attributes in a relation *R*, one can derive several properties of functional dependencies. Among the most important are Armstrong's axioms, which are used in database normalization:

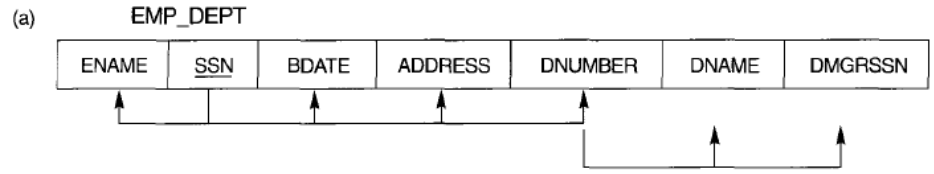
- **Subset Property** (Axiom of Reflexivity): If *Y* is a subset of *X*, then $X \rightarrow Y$
- **Augmentation** (Axiom of Augmentation): If $X \rightarrow Y$, then $XZ \rightarrow YZ$
- **Transitivity** (Axiom of Transitivity): If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

From these rules, we can derive these secondary rules:

- **Union**: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
- **Decomposition**: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
- **Pseudotransitivity**: If $X \rightarrow Y$ and $WY \rightarrow Z$, then $XW \rightarrow Z$

Equivalent sets of functional dependencies are called covers of each other. Every set of functional dependencies has a canonical cover

For example, suppose that we specify the following set F of obvious functional dependencies on the relation schema of Figure a:



F= {SSN → {ENAME, BDATE, ADDRESS, DNUMBER},

DNUMBER → {DNAME, DMGRSSN}}

Some of the additional functional dependencies that we can *infer* from F are the following:

SSN → {DNAME, DMGRSSN}

SSN →SSN

DNUMBER →DNAME

Database normalization

In the field of relational database design, **normalization** is a systematic way of ensuring that a database structure is suitable for general-purpose querying and free of certain undesirable characteristics—insertion, update, and deletion anomalies—that could lead to a loss of data integrity.

Background to normalization: definitions

Functional dependency

In a given table, an attribute *Y* is said to have a functional dependency on a set of attributes *X* (written *X* → *Y*) if and only if each *X* value is associated with precisely one *Y* value. For example, in an "Employee" table that includes the attributes "Employee ID" and "Employee Date of Birth", the functional dependency {Employee ID} → {Employee Date of Birth} would hold.

Trivial functional dependency

A trivial functional dependency is a functional dependency of an attribute on a superset of itself. {Employee ID, Employee Address} → {Employee Address} is trivial, as is {Employee Address} → {Employee Address}.

Full functional dependency

An attribute is fully functionally dependent on a set of attributes X if it is

- functionally dependent on X, and

- not functionally dependent on any proper subset of X. {Employee Address} has a functional dependency on {Employee ID, Skill}, but not a *full* functional dependency, because it is also dependent on {Employee ID}.

Transitive dependency

A transitive dependency is an indirect functional dependency, one in which $X \rightarrow Z$ only by virtue of $X \rightarrow Y$ and $Y \rightarrow Z$.

Multivalued dependency

A multivalued dependency is a constraint according to which the presence of certain rows in a table implies the presence of certain other rows.

Join dependency

A table *T* is subject to a join dependency if *T* can always be recreated by joining multiple tables each having a subset of the attributes of *T*.

Superkey

A superkey is an attribute or set of attributes that uniquely identifies rows within a table; in other words, two distinct rows are always guaranteed to have distinct superkeys. {Employee ID, Employee Address, Skill} would be a superkey for the "Employees' Skills" table; {Employee ID, Skill} would also be a superkey.

Candidate key

A candidate key is a minimal superkey, that is, a superkey for which we can say that no proper subset of it is also a superkey. {Employee Id, Skill} would be a candidate key for the "Employees' Skills" table.

Non-prime attribute

A non-prime attribute is an attribute that does not occur in any candidate key. Employee Address would be a non-prime attribute in the "Employees' Skills" table.

Primary key

Most DBMSs require a table to be defined as having a single unique key, rather than a number of possible unique keys. A primary key is a key which the database designer has designated for this purpose.

Normal forms

The **normal forms** (abbrev. **NF**) of relational database theory provide criteria for determining a table's degree of vulnerability to logical inconsistencies and anomalies. The higher the normal form applicable to a table, the less vulnerable it is to inconsistencies and anomalies. Each table has a "**highest normal form**" (**HNF**): by definition, a table always meets the requirements of its HNF and of all normal forms lower than its HNF; also by definition, a table fails to meet the requirements of any normal form higher than its HNF

The normal forms are applicable to individual tables; to say that an entire database is in normal form *n* is to say that all of its tables are in normal form *n*.

First normal form

First normal form (**1NF or Minimal Form**) is a normal form used in database normalization. A relational database table that adheres to 1NF is one that meets a certain minimum set of criteria. These criteria are basically concerned with ensuring that the table is a faithful representation of a **relation** and that it is free of **repeating groups.**

A table is in 1NF if and only if it is "isomorphic to some relation", which means, specifically, that it satisfies the following five conditions:

1. There's no top-to-bottom ordering to the rows.
2. There's no left-to-right ordering to the columns.
3. There are no duplicate rows.
4. Every row-and-column intersection contains exactly one value from the applicable domain (and nothing else).
5. All columns are regular [i.e. rows have no hidden components such as row IDs, object IDs, or hidden timestamps].

Violation of any of these conditions would mean that the table is not strictly relational, and therefore that it is not in 1NF.

Examples of tables (or views) that would not meet this definition of 1NF are:

- A table that lacks a unique key. Such a table would be able to accommodate duplicate rows, in violation of condition 3.
- A view whose definition mandates that results be returned in a particular order, so that the row-ordering is an intrinsic and meaningful aspect of the view. This violates condition 1. The tuples in true relations are not ordered with respect to each other.
- A table with at least one nullable attribute. A nullable attribute would be in violation of condition 4, which requires every field to contain exactly one value from its column's domain.

Example

Suppose a novice designer wishes to record the names and telephone numbers of customers. He defines a customer table which looks like this:

Customer			
Customer ID	**First Name**	**Surname**	**Telephone Number**
123	Robert	Ingram	555-861-2025
456	Jane	Wright	555-403-1659
789	Maria	Fernandez	555-808-9633

The designer then becomes aware of a requirement to record **multiple** telephone numbers for some customers. He reasons that the simplest way of doing this is to allow the "Telephone Number" field in any given record to contain more than one value:

Customer			
Customer ID	**First Name**	**Surname**	**Telephone Number**
123	Robert	Ingram	555-861-2025
456	Jane	Wright	555-403-1659 555-776-4100
789	Maria	Fernandez	555-808-9633

Assuming, however, that the Telephone Number column is defined on some Telephone Number-like constraint (e.g. strings of 12 characters in length), the representation above is not in 1Nf as it prevents a single field from containing more than one value from its column's domain.

The designer might attempt to get around this restriction by defining multiple Telephone Number columns:

Customer					
Customer ID	**First Name**	**Surname**	**Tel. No. 1**	**Tel. No. 2**	**Tel. No. 3**
123	Robert	Ingram	555-861-2025		
456	Jane	Wright	555-403-1659	555-776-4100	555-403-1659
789	Maria	Fernandez	555-808-9633		

This representation, however, makes use of nullable columns, and therefore does not conform to Date's definition of 1NF and causes logical problems. These problems include:

- Difficulty in querying the table. Answering such questions as "Which customers have telephone number *X*?"
- Inability to enforce uniqueness of Customer-to-Telephone Number links through the RDBMS. Customer 789 might mistakenly be given a Tel. No. 2 value that is exactly the same as her Tel. No. 1 value.
- Restriction of the number of telephone numbers per customer to three. If a customer with four telephone numbers comes along, we are constrained to record only three and leave the fourth unrecorded. This means that the database design is imposing constraints on the business process, rather than (as should ideally be the case) vice-versa.

A design that is unambiguously in 1NF makes use of two tables: a Customer Name table and a Customer Telephone Number table.

Customer Name		
Customer ID	**First Name**	**Surname**
123	Robert	Ingram
456	Jane	Wright
789	Maria	Fernandez

Customer Telephone Number	
Customer ID	**Telephone Number**
123	555-861-2025
456	555-403-1659
456	555-776-4100
789	555-808-9633

Repeating groups of telephone numbers do not occur in this design. Instead, each Customer-to-Telephone Number link appears on its own record.

Second normal form

A table that is in first normal form (1NF) must meet additional criteria if it is to qualify for second normal form. Specifically: a 1NF table is in 2NF if and only if, given any candidate key K and any attribute A that is not a constituent of a candidate key, A depends upon the whole of K rather than just a part of it.

In slightly more formal terms: a 1NF table is in 2NF if and only if all its non prime attributes are functionally dependent on the whole of a candidate key. (A non prime attribute is one that does not belong to any candidate key.)

Example

Consider a table describing employees' skills:

Employees' Skills		
Employee	**Skill**	**Current Work Location**
Jones	Typing	114 Main Street
Jones	Shorthand	114 Main Street
Jones	Whittling	114 Main Street
Bravo	Light Cleaning	73 Industrial Way
Ellis	Alchemy	73 Industrial Way
Ellis	Flying	73 Industrial Way
Harrison	Light Cleaning	73 Industrial Way

Neither {Employee} nor {Skill} is a candidate key for the table. This is because a given Employee might need to appear more than once (he might have multiple Skills), and a given Skill might need to appear

more than once (it might be possessed by multiple Employees). Only the composite key {Employee, Skill} qualifies as a candidate key for the table.

The remaining attribute, Current Work Location, is dependent on only part of the candidate key, namely Employee. Therefore the table is not in 2NF. Note the redundancy in the way Current Work Locations are represented: we are told three times that Jones works at 114 Main Street, and twice that Ellis works at 73 Industrial Way. This redundancy makes the table vulnerable to update anomalies: it is, for example, possible to update Jones' work location on his "Typing" and "Shorthand" records and not update his "Whittling" record. The resulting data would imply contradictory answers to the question "What is Jones' current work location?"

A 2NF alternative to this design would represent the same information in two tables: an "Employees" table with candidate key {Employee}, and an "Employees' Skills" table with candidate key {Employee, Skill}:

Employees	
Employee	**Current Work Location**
Jones	114 Main Street
Bravo	73 Industrial Way
Ellis	73 Industrial Way
Harrison	73 Industrial Way

Employees' Skills	
Employee	**Skill**
Jones	Typing
Jones	Shorthand
Jones	Whittling
Bravo	Light Cleaning
Ellis	Alchemy
Ellis	Flying
Harrison	Light Cleaning

Neither of these tables can suffer from update anomalies.

Third normal form

The **third normal form** normal form used in database normalization. 3NF was originally defined by E.F. Codd in 1971. Codd's definition states that a table is in 3NF if and only if both of the following conditions hold:

- The relation R (table) is in second normal form (**2NF**)
- Every non-prime attribute of R is non-transitively dependent (i.e. directly dependent) on every candidate key of R.

A **non-prime attribute** of R is an attribute that does not belong to any candidate key of R. A transitive dependency is a functional dependency in which $X \rightarrow Z$ (X determines Z) indirectly, by virtue of $X \rightarrow Y$ and $Y \rightarrow Z$ (where it is not the case that $Y \rightarrow X$).

Example

An example of a 2NF table that fails to meet the requirements of 3NF is:

Tournament Winners			
Tournament	**Year**	**Winner**	**Winner Date of Birth**
Indiana Invitational	1998	Al Fredrickson	21 July 1975
Cleveland Open	1999	Bob Albertson	28 September 1968
Des Moines Masters	1999	Al Fredrickson	21 July 1975
Indiana Invitational	1999	Chip Masterson	14 March 1977

Because each row in the table needs to tell us who won a particular Tournament in a particular Year, the composite key {Tournament, Year} is a minimal set of attributes guaranteed to uniquely identify a row. That is, {Tournament, Year} is a candidate key for the table.

The breach of 3NF occurs because the non-prime attribute Winner Date of Birth is transitively dependent on the candidate key {Tournament, Year} via the non-prime attribute Winner. The fact that Winner Date of Birth is functionally dependent on Winner makes the table vulnerable to logical inconsistencies, as there is nothing to stop the same person from being shown with different dates of birth on different records.

In order to express the same facts without violating 3NF, it is necessary to split the table into two:

Tournament Winners		
Tournament	**Year**	**Winner**
Indiana Invitational	1998	Al Fredrickson
Cleveland Open	1999	Bob Albertson
Des Moines Masters	1999	Al Fredrickson
Indiana Invitational	1999	Chip Masterson

Player Dates of Birth	
Player	**Date of Birth**
Chip Masterson	14 March 1977
Al Fredrickson	21 July 1975
Bob Albertson	28 September 1968

Update anomalies cannot occur in these tables, which are both in 3NF.

Boyce-Codd normal form

Boyce-Codd normal form (or **BCNF** or **3.5NF**) is a normal form used in database normalization. It is a slightly stronger version of the third normal form (3NF). A table is in Boyce-Codd normal form if and only if for every one of its non-trivial [dependencies] $X \rightarrow Y$, X is a superkey—that is, X is either a candidate key or a superset thereof.

BCNF was developed in 1974 by Raymond F. Boyce and Edgar F. Codd to address certain types of anomaly not dealt with by 3NF as originally defined.

Only in rare cases does a 3NF table not meet the requirements of BCNF. Depending on what its functional dependencies are, a 3NF table with two or more overlapping candidate keys may or may not be in BCNF.

An example of a 3NF table that does not meet BCNF is:

Today's Court Bookings			
Court	**Start Time**	**End Time**	**Rate Type**
1	09:30	10:30	SAVER
1	11:00	12:00	SAVER
1	14:00	15:30	STANDARD
2	10:00	11:30	PREMIUM-B
2	11:30	13:30	PREMIUM-B
2	15:00	16:30	PREMIUM-A

1. Each row in the table represents a court booking at a tennis club that has one hard court (Court 1) and one grass court (Court 2)
2. A booking is defined by its Court and the period for which the Court is reserved
3. Additionally, each booking has a Rate Type associated with it. There are four distinct rate types:
    - SAVER, for Court 1 bookings made by members
    - STANDARD, for Court 1 bookings made by non-members
    - PREMIUM-A, for Court 2 bookings made by members
    - PREMIUM-B, for Court 2 bookings made by non-

The table's candidate keys are:

- {Court, Start Time}
- {Court, End Time}
- {Rate Type, Start Time}
- {Rate Type, End Time}

In the Today's Court Bookings table, there are no non prime attributes: that is, all attributes belong to candidate keys. Therefore the table adheres to both 2NF and 3NF.

The table does not adhere to BCNF. This is because of the dependency Rate Type → Court, in which the determining attribute (Rate Type) is neither a candidate key nor a superset of a candidate key.

Any table that falls short of BCNF will be vulnerable to logical inconsistencies. In this example, enforcing the candidate keys will not ensure that the dependency Rate Type → Court is respected. There is, for instance, nothing to stop us from assigning a PREMIUM A Rate Type to a Court 1 booking as well as a Court 2 booking—a clear contradiction, as a Rate Type should only ever apply to a single Court.

The design can be amended so that it meets BCNF:

Rate Types		
Rate Type	**Court**	**Member Flag**
SAVER	1	Yes
STANDARD	1	No
PREMIUM-A	2	Yes
PREMIUM-B	2	No

Today's Bookings			
Court	**Start Time**	**End Time**	**Member Flag**
1	09:30	10:30	Yes
1	11:00	12:00	Yes
1	14:00	15:30	No
2	10:00	11:30	No
2	11:30	13:30	No
2	15:00	16:30	Yes

The candidate keys for the Rate Types table are {Rate Type} and {Court, Member Flag}; the candidate keys for the Today's Bookings table are {Court, Start Time} and {Court, End Time}. Both tables are in BCNF. Having one Rate Type associated with two different Courts is now impossible, so the anomaly affecting the original table has been eliminated.