CHAPTER 4
Database Normalization

Section 1. Database Normalization - Why Is It Important?

Database normalization is the process of structuring one or more relations in a relational **database** schema in accordance with a series of so-called *normal forms* in order to reduce the amount of redundant data that is, or would be, stored in a volatile database (that is, one that regularly has tuples being added, deleted, or changed). As a consequence this will improve the integrity of that data. We begin by illustrating what we mean by the data redundancy and data integrity with an example.

Suppose a database scheme for a manufacturer consists of the following relation schema, giving information about parts and suppliers

```
    PARTSSUPPLIED(PartNum,PartDesc,SupplierNo,SupplierName, SupplierLocation, Price)
```

As indicated by the underlining, the primary key for this relation is the composite key `(Part#,Supplier#)`.

Suppose the following table represents a sample of the data in the relation:

PartNum	PartDesc	SupplierNum	SupplierName	SupplierLocation	Price
1	aaa	1000	JONES	N. CHARLESTON	20
1	aaa	1500	ABC	CHARLESTON	28
1	aaa	2050	XYZ	COLUMBIA	22
1	aaa	1900	P&H	ATLANTA	30
2	bbb	3100	ALLEN	ATLANTA	520
2	bbb	1000	JONES	NORTH CHARLESTON	500
2	bbb	2050	XYZ	COLUMBIA	590
3	ccc	2050	XYZ	COLUMBIA	1000
4	ddd	1000	JONES'	N CHARLESTON	80
4	ddd	3100	ALLEN	ATLANTA	90
4	ddd	1900	P&H	ATLANTA	95
5	eee	1500	ABC	CHARLESTON	160
5	eee	1000	JONES	N. CHARLESTON	140
5	eee	8156	GJP	MT. PLEASANT	190

We can now see several problems with this scheme:

1. (*Redundancy*) The values for `PartDesc`, `SupplierName`, and `SupplierLocation` are repeated each time for each `Part`Num and `Supplier`Num. One consequence is wasting storage, but more significantly, the "repeated" values may not always be the same.

2. (*Update Anomalies*) As a consequence of the redundancy shown above, if a Supplier changes Name or Location, or if a Part Description is changed, then all affected tuples in the relation (or database if such redundancy is present in other relations) would have to be changed –– leading to the possibility that the name or location for a supplier, or the description for a part in one tuple may be changed, but not another. Consequently we may not have a unique name or location for each supplier or a unique name for each part, when we feel intuitively that we should.

   Example: Suppose in relation above that supplier JONES changes location to SAVANNAH and renames itself SMITH, then the following update anomalies are possible

:

PartNum	**PartDesc**	**SupplierNum**	**SupplierName**	**SupplierLocation**	**Price**
1	aaa	1000	SMITH	SAVANNAH	20
1	aaa	1500	ABC	CHARLESTON	28
1	aaa	2050	XYZ	COLUMBIA	22
1	aaa	1900	P&H	ATLANTA	30
2	bbb	3100	ALLEN	ATLANTA	520
2	bbb	1000	SMITH	NORTH CHARLESTON	500
2	bbb	2050	XYZ	COLUMBIA	590
3	ccc	2050	XYZ	COLUMBIA	1000
4	ddd	1000	JONES'	N CHARLESTON	80
4	ddd	3100	ALLEN	ATLANTA	90
4	ddd	1900	P&H	ATLANTA	95
5	eee	1500	ABC	CHARLESTON	160
5	eee	1000	SMITH	SAVANNAH	140
5	eee	8156	GJP	MT. PLEASANT	190

3. *(Insertion Anomalies*) According to our relation schema, and its compound primary key, we cannot enter any information about a supplier into our database unless the supplier supplies at least one part. Thus, if we had information a potential supplier, say WAM from SALZBURG (whom we would assign supplier number 1756), we cannot enter this information until we have been supplied with one of their parts.

   - Since `PartNum` and `SupplierNum` together form a key for the relation, we would need to have a SupplierNum value that represents "no supplier yet, and we can insert **null** values `PartDesc`, and `Price`, but then, when we enter an actual item for the supplier will someone remember to delete the tuple with the null values? Or if we delete the only item supplied will we know to reinsert a tuple with the "No supplier yet" number?

4. (*Deletion Anomalies*) If we delete all of the tuples with a given `SupplierNum` (say 8156) we lose all information about that supplier.

In the database/relation scheme used in the above example, the problems cited can be eliminated by using the following database scheme instead:

```
PART(PartNum,PartDesc)
SUPPLIER(SupplierNumSupplierName,SupplierLocation)
PARTSSUPPLIED(PartNum,SupplierNum,Price)
```

There is a disadvantage to the above decomposition, however. To find the suppliers of part 1, for example, we must use a query that joins the two tables, which can be time-expensive for very large databases. With the single relation scheme we could simply do a selection and a projection. Nevertheless, for databases which are volatile the advantages of the multi-relation scheme generally outweigh those of the single relation scheme.

What we now undertake is to describe how to specify a structure that reduces the tendency of the database to allow insertion, deletion, and modification anomalies. The concept underlying this structure is that of a functional dependency.

Section 2. Functional Dependency

As we noted in the first sentence of the previous section, **normalization** is a process used in the design of a relational database to provide a systematic way of ensuring a minimal amount of data redundancy, while at the same time avoiding the various types of anomalies discussed above.

Normalization gets its name from its use of *normal forms*, where a relation is said to be in a particular normal form if it satisfies a relevant set of conditions. In the rest of this section we shall discuss the types of normal forms known as ***Second Normal Form*** or 2NF,and ***Third Normal Form*** or 3NF. There is also a First Normal Form (1NF), and Boyce-Codd Normal Form (or BCNF),Fourth Normal Form (or 4NF) and a Fifth Normal Form (or 5NF, or PJNF -- for project join normal form). We shall not discuss these forms, however, since they are not germane to our needs.

Since the normal forms we consider assume a relation in first normal form, we give its definition for the sake of completion, but do not go beyond this.

First Normal Form: A relation is in **first normal form**(1NF) if the values allowed for each attribute are atomic (that is, they are not a set of values, or values with labelled components)

Example: See handout 4-2-1NFExample

Functional Dependency

The normal forms which we will consider are aimed at removing some of the conditions which lead to the anomalies we described at the beginning of this section. In order to describe these conditions satisfactorily, however, we must begin with the notion of a "functional dependency."

Let **R** = **R(A_1,..., A_n)** be a relation scheme and let X and Y be disjoint subsets of $\{A_1,...,A_n\}$. We say that *Y is functionally dependent on X* (or that *X functionally determines Y*), denoted $X \rightarrow Y$, if for arbitrary tuples t_1 and t_2 of **R**, it is impossible for t_1 and t_2 to agree on the values in the X attributes and to differ on one or more of the values of the Y attributes. X is called a *determinant* of Y.

Example: Consider the relation scheme

```
STUDENT(STID, StName, Major, Credits, Status, SSN)
```

together with the following constraints:

- Every student has a **STID** value and a **SSN** value that is unique for that student.
- Every student has at most one major.
- Names are not unique.
- **Credits** refers to credit hours completed.
- **Status** refers to the student's year in school - freshman (FR), sophomore (SO), etc.

Based on these assumptions we could identify (or assign) the following functional dependencies in this relation scheme:

```
STID → StName, Major, Credits, Status, SSN
SSN → STID, StName, Major, Credits, Status
Credits → Status
```

Let **R** = **R(A_1,..., A_n)** be a relation scheme and let X and Y be disjoint subsets of $\{A_1,...,A_n\}$. We say that *Y is fully functionally dependent on X* if Y is functionally dependent on X and is not functionally on a subset of the attributes comprising X.

Example: Consider the following relation scheme

```
CLASS(CourseNum, STID, StName, FACID, Sched, Room, Grade)
```

where

- **CourseNum** includes the department, course, and section.
- Only one faculty member teaches a given section of a course (that is, no team teaching).
- **Sched** gives the meeting times and days (as one string).
- **Room** gives the building and room (again as one string).

We can identify the following dependencies:

```
CourseNum, STID → StName, FACID, Sched, Room, Grade
CourseNum → FACID, Sched, Room
STID → StName
```

Here **StName**, **FACID**, **Sched**, and **Room** are functionally dependent on C**ourseNum**, **STID** but are not fully functionally dependent on it. **Grade** is fully functionally dependent on **CourseNum**,**STID** however.

It is important to note that a functional dependency and full functional dependency are properties of the semantics of a relation R, and are not properties of a particular instance of R. In particular, we use our understanding of the associations we want to hold between attributes of R to specify the functional dependencies that must exist for all its instances and from these to also ascertain full functional dependencies. ***A functional dependency cannot be deduced from instances of R, but must be defined explicitly by someone who knows the semantics of the attributes of R***. On the other hand it only requires a single counterexample to disprove a functional dependency, or a full functional dependency.

Second Normal Form (2NF)

A relation is in ***second normal form*** (2NF) if it is in first normal form and all the non-key attributes are fully functionally dependent on the primary key[1].

- It is important to play close attention to the role played by primary key attributes and non-key attributes in this definition. The only time one needs to be concerned about a relation not being in 2NF is when the primary key is composite.

A relation which is not in 2NF is prone to the anomalies described at the beginning of this section.

Examples:

A relation which is not in 2NF can be transformed into a set of relations which are in 2NF without losing information from the original relation by identifying each non-full functional dependency and forming projections to remove the attributes that depend on the determinants of each of the non-full dependencies. The determinant and their dependent attributes are placed in a separate relation. The determinant is not removed from the original relation, however, but rather become foreign keys in that relation.

Example:

One can reconstruct the original relation by taking the natural join of the relations created from it.

[1] There is a more general definition of 2NF to exclude partial dependencies of so called non-prime attributes on any candidate key. The definition we adopt is sufficient for our purposes, however.

Transitive Dependence and Third Normal Form

Although Second Normal Form can eliminate transaction anomalies that result from attributes not being fully functionally dependent on the key of a relation, 2NF by itself will not guarantee that transaction anomalies will not take place. Consider, for example, the following relation scheme

```
TEACHES(Course,Prof,Room,RoomCap)
```

where the domain of the attribute **Course** is the courses offered by a department in a particular semester, the domain of the attribute **Prof** is the faculty members of the department, the domain of **Room** is the rooms assigned to that department for teaching its courses, and the domain of **RoomCap** is an integer indicating the seating capacity of the room. We assume that only one professor teaches a given course and that a given course always meets in the same room., although the same room may be used for more than one course. Note that since **Course** alone is the key for **TEACHES**, the relation is automatically in 2NF. This relation however is still prone to insertion, deletion, and update anomalies of the type we saw earlier. To see this, suppose the following table represents an instance of this relation:

Course	**Prof**	**Room**	**RoomCap**
353	Smith	A532	45
351	Smith	C320	100
355	Clark	H940	400
456	Turner	B278	50
459	Jamieson	D110	50
480	Clark	A532	45

Note, however:

- We cannot insert the information that A560 has capacity 50 until a course is assigned to that room.
- If we delete the only course scheduled for a given room, say course 355 in the above instance, we lose information about the capacity of that room.
- If the capacity of a room changes, say the capacity of room A532 is decreased to 40, then the database will be susceptible to update anomalies.

The reason why 2NF could not prevent these anomalies from possibly arising is that 2NF only addresses functional dependencies between non-key attributes and the key. Let us examine the dependencies which exist in the **TEACHES** scheme. Obviously we have the dependencies

Course → Prof, **Room**, **RoomCap**

but we also have the following dependency

Room → RoomCap

which does not involve the primary key of **TEACHES**. If we can eliminate such dependencies from the relations in our database scheme then we might be able to eliminate transaction anomalies from the database. It seems that what we need is a general way to classify a relation scheme so that functional dependencies "away from the key" are not permitted.

One can show that for a relation with attributes X, Y, and Z, if $X \rightarrow Y$, and $Y \rightarrow Z$, then $X \rightarrow Z$ - a phenomenon known as *transitive dependency*. In the above example, in addition to the (full) functional dependency of the (nonkey) attribute **RoomCap** on **Courses** (the relation scheme's key), there is a also transitive dependency of **RoomCap** on **Course** through **Room**.

A relation scheme is in ***third normal form*** (3NF) if it is in 2NF and no non-key attribute is transitively dependent on the key.

- "All non-key attributes depend on the key, the whole key, and nothing but the key. "

A relation which is 2NF but not in 3NF can be transformed into an equivalent set of 3NF relations by finding any non-key attributes which are transitively dependent on the key and placing them and their determinants in a new relation.

Example: Given the relation scheme **TEACHES** above, we can decompose it into the following pair of 3NF relation schemes:

```
COURSE(CourseNum,Prof,RoomNum)
ROOM(RoomNum,RoomCap)
```

Example:

Denormalization

One of the advantages of normalizing a relational database schema is that data duplication is minimized, reducing the possibilities of update anomalies arising. Sometimes, however, it is appropriate to intentionally allow duplicate data; usually this is to improve the performance of certain commonly occurring queries. When this is desired, one may choose not to further normalize a relation, or may even elect to recombine some normalized relations to form a relation that is not normalized (a process known as *denormalization*). Denormalization can be especially beneficial when one is working with archival data (such as in a data warehouses) where data modifications are rare.

Example: Consider the relational scheme

```
CONTRACTS(CID, SuppID, ProjID, DeptID, PartID, Qty, Value)
DEPARTMENTS(DID, Budget, AnnualRpt)
PARTS(PID, Cost)
PROJECTS(JID, Manager)
SUPPLIERS(SID, Location)
```

The semantics of the **CONTRACTS** relations is such that a contract with identifier **CID** is an agreement that a supplier (with **SuppID** equal to some **SID**) will supply **Qty** items of a part (with **PartID** equal to some **PID**) to a project (with **ProjID** equal to some **JID**) associated with a department (with **DeptID** equal to some **DID**). The attribute **Value** represents the total value of the contract.

It is also understood that there will not be two distinct contracts in which the same project buys the same part. This constraint is represented by the functional dependency **ProjID+PartID → CID**.

Note that there are two candidate keys for **CONTRACTS** -- **CID** and **ProjID+PartID.** There are no transitive dependencies in **CONTRACTS**, whence **CONTRACTS** is in 3NF.

Now suppose the data in the database is essentially frozen and suppose also that a frequent query is to check that the value of a contract is less than the budget of the contracting department. Rather than execute a join between **CONTRACTS** and **DEPARTMENTS** to implement this query we may want to add an attribute **Budget** to **CONTRACTS** and in so doing introduce the dependency **DeptID → Budget.** This would mean that **CONTRACTS** is no longer in 3NF and may have a significant amount of data redundancy. Nevertheless if having the motivating query execute quickly is a high priority then this may outweigh the consequences of the data redundancy, especially if we will not be altering the data.