The Relational Model

The most prevalent model in databases is the relational database model. It is so successful because:

- It corresponds with natural notions for modeling (the idea of writing data in tables, with rows and columns)
- It is based on set theory, and the concept of relations, which means things can be proved about its power
- We have invented efficient ways to implement it and optimize its performance.

1. Informal introduction

The relational model closely correspond with how we normally represent data. For example, imagine you are asked to create a class roster, containing the id, name and gender of students in a class. You'd probably come up with something like this:

Id	Name	Gender
12345	Orlando Karam	M
34567	Lina Colli	F
...	...	...

There are a few issues that are not clear from the table, but I hope you'd agree:

- We wouldn't put the *same* row twice; there can be no duplicates
- The order of the rows is not terribly important (although we would choose a particular order for convenience, if the rows were in a different one, we would still say it is the same table)
- The order of the *columns* is not terribly important either
- You'd expect the same kind of values to go in each column; that is, you wouldn't expect the value 12345 to appear in the gender column.

These issues will become important when we formally define a relation (which is our fancy name for a table). Notice that we may leave some cells blank; in the relational model we use a special value called null for blank cells.

Now, lets assume you are asked to add emails to the 'database', and we want to keep as many emails as a person would give us. We could try a couple of different modifications, that are not entirely satisfactory:

- We could add several fields, email1, email2 etc, yielding a table like this:

Id	Name	Gender	email1	email2
12345	Orlando Karam	M		
34567	Lina Colli	F		
...	...	...		

There are a couple of problems with this solution; first, we can only add a limited number of emails; also, we would be wasting space for people with fewer emails, and there may be data integrity issues (people filling email2 but not email1, etc)

- We could add one field, say emails, and give the field some internal structure; for example, ask people to list all their emails, separated by commas. This could yield to integrity problems again (some people separate with semicolon), and it makes the data opaque to the DBMS (and to programmers), which makes validation and use of the data harder.
- We could add one field, email and list all emails down from the person they belong to, as follows:

Id	Name	Gender	emails
12345	Orlando Karam	M	ok@ok.com
			ok@spsu.edu
			ok@coldmail.com
34567	Lina Colli	F	
...	...	...	

Here the problem is that the order of the rows becomes important ! Now if we sort the rows to view them differently, we mix up the emails of different people. Although this *could* work for a simple spreadsheet, it wouldn't work for a large database.

- We could add one field, emails, and *copy* the other data for the row, as follows:

Id	Name	Gender	emails
12345	Orlando Karam	M	ok@ok.com
12345	Orlando Karam	M	ok@spsu.edu
12345	Orlando Karam	M	ok@coldmail.com
34567	Lina Colli	F	
...	...	...	

Now the problem is that we are repeating the same information many times; if we realized we had mistyped Orlando's name, we would need to change it in 3 different places. We will later talk about normalization, which deals with solving this issue.

Now, if we think hard, we would probably come up with the idea of having *two* tables, one containing the person's data and another containing the emails; basically adding the following table to our original one:

Id	emails
12345	ok@ok.com
12345	ok@spsu.edu
12345	ok@coldmail.com
34567	lc@lc.com
...	...

Notice this requires two things:

- There can be no two people in the original table with the same id (this ties with the concept of primary key)

- Any values in the id field of the emails table needs to come from the id's of the original table (this is the idea of a foreign key)

This last solution works great as long as we have an easy way to go from one table to the other; the relational model provides this through the *join* operation.

2. Formalizing the model

We define a **tuple** as a set of names, with a value associated with each name; we use the special value null to associate with a name when there is no other reasonable value for it (so we can get tuples with the same set of attributes). We define a **relation** as a set of tuples, all of the same type (that is, having the same set of names). A relation has two parts:

- a *schema* (or *heading* , *or intension)* a set of names, with a *type* or *domain* associated with each name (the domain is the set of all possible values the attribute could have associated in the body; basically the data type for the attribute). Notice that the special value null is, by default, a member of every basic domain.
- a *body* (or *extension*) which is a set of tuples (*or rows*)*,* each of them having the same attribute names as the schema, and with values for each attribute from the corresponding domain.

Relations are associations between *values*, and as such, they are closely related to the mathematical concepts of relations and functions.

Remember that a *set* is an unordered collection with no duplicates; that is, the order is not important, and there can be no duplicates; so, from the definition of a set and that of a relation we can infer the following properties of relations:

1. There are no duplicate tuples (or rows)
2. Tuples are unordered; that is, the order of the rows doesn't matter
3. Attributes are unordered; that is, the order of the columns doesn't matter
4. The attribute names are unique within a relation[1]; this implies there is only one value per attribute for each tuple.

Keys

Since relations are sets of tuples, they can't contain duplicates; however, we normally have stronger constraints; in many relations there are sets of attributes for which there can be no duplicates; we normally call those sets *keys*. More formally, we have the following definitions:

- A **superkey** is any set of attributes that is guaranteed to have unique values for each row in a relation.
- A **candidate key** is a minimal superkey; that is, a superkey that, if we take out any attribute, ceases to be a superkey.
- A **prime attribute** is any attribute which is part of a candidate key.
- A **foreign key** is a set of attributes in a relation whose values are taken from the set of values from a candidate key (usually the primary key) of some other relation.

[1] Notice that many people extend this requirement to the whole database, and require that the name of each attribute is unique across the whole database schema.

- The **primary key** of a relation is a candidate key that we choose to be used for foreign keys from other relations.

Notice that in many cases a key is composed of just one attribute. Also, the set of all attributes in a relation is always a superkey, which implies that every relation has at least one candidate key. Although it is not a requirement to choose a primary key, we assume that one will always be chosen, and that all foreign key references to a given relation will be through that primary key.

Keys are used to guarantee the integrity of the data. We have the following integrity rules:

- The **entity integrity rule** stipulates that no prime attribute can be null; in particular, no attribute which is part of the primary key can be null.
- The **referential integrity rule** stipulates that for foreign keys, either the attributes match the value of some row in the corresponding relation, or the foreign key is null.

3. Relational Schema Diagrams

It is oftentimes convenient to represent a relational database schema in a diagram (later on, we will use SQL to actually create the database). The conventions are relatively simple:

- For each table, we write its name, and then all its fields (we usually put the borders around the field names).
- The fields that form the primary key are underlined
- All foreign key relationships among tables are marked with an arrow that goes from the foreign key to the primary key. Optionally, you can *also* dash-underline the foreign key, as Hoffer's oftentimes does. Notice you *always* have to put the arrow.

Probably the easiest way to create those diagrams is to use a modern word processor that allows you to create tables and to draw arrows on a page. In particular, recent versions of OpenOffice.org and MS Word allow you to do that (this document was typed with OpenOffice.org).

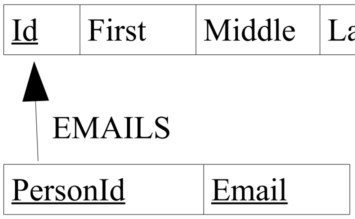
So, if we have just one table, called Person we can do:

PERSON

Id	First	Middle	Last	Age

And if we have related tables, we just draw arrows from the foreign keys to the primary keys. For example, if Person had a multi-valued emails attribute, we would need to use two tables, as follows:

PERSON



Id	First	Middle	Last	Age

EMAILS

PersonId	Email

4. Normalization

When creating our relational schema, we might end up with badly designed tables, if we are not careful. Normalization is a technique to validate our relational design and make sure we are not doing unnecessary duplication. Intuitively, the most important rules is that **your table corresponds to only one fact**, but normalization theory allows us to precisely define what we mean by that.

After we have normalized our design, we are guaranteed our relations are **well-formed**, that is, they do not lead to **anomalies**.

An anomaly would be a situation in which we cannot do a straight insert update or delete operation due to the design of our tables. For example, look at the following table:

Course Num	Course Title	Section Num	Term	DoW	Time
CS3153	Databases	1	F07	TR	4:30p
CS3153	Databases	900	F07		Online
CS1301	Intro Prog	1	F07	MW	11:00a

This table is badly designed, since it talks about two things; the courses (with their titles) and the sections for the course (with term, day of week etc). It leads to the following anomalies:

- **Insertion anomalies** If we insert a new section of CS3153, we also need to make sure that the title is recorded as Databases
- **Update anomalies** If we were to change the title of the CS3153 course, we need to change more than one row
- **Deletion anomalies** If we delete the CS1301 section, we are also eliminating from our database the fact that the title for CS1301 is Intro Prog

The solution of course, would be to break the table into two tables, as follows:

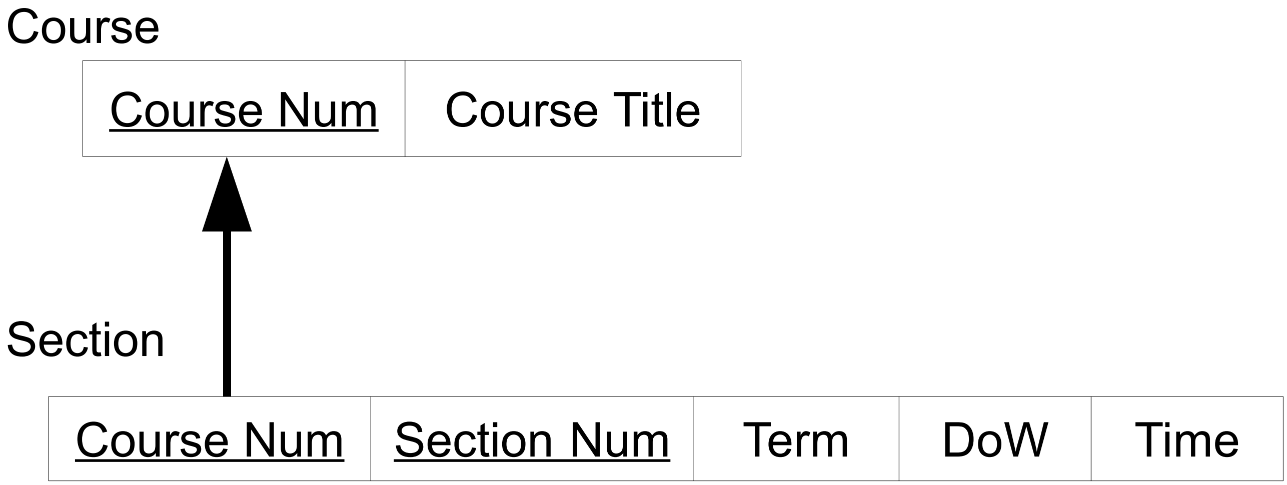


Figure 1: Relational schema for table 1, courses and titles

You can verify this design eliminates the anomalies mentioned above.

So, we know there are bad designs and that we can solve those problems; now we need a theory that helps us detect those bad designs and fix them. This is Normalization theory.

Functional Dependencies

Normalization theory (up to 3rd normal form, 3NF) is based on the concept of functional dependencies . A functional dependency (FD) is a particular kind of constraint on the allowed values of a relation. We represent an FD with the notation A -> B, where both A and B are **sets** of attributes in a relation. We say that an FD A->B **holds** on a relation, if for any two tuples that have the same value of A, they must necessarily have the same value of B.

Notice that at both sides of an FD we can put **sets** of attributes. For notational convenience, we do not use anything special to denote the sets, and just separate attributes with commas if there is more than one. If we were to use traditional set notation, FDs would look like this: {A}->{B}; we are just eliminating the braces.

The set of attributes in the left-hand side of an FD is called the **determinant**.

Notice that a functional dependency is a generalization of the concept of candidate key. If a set of attributes is a key for a table, then there cannot be two rows with the same value for those attributes. So on any table, all attributes are dependent on the primary key. So, on the relational schema on figure 1, Course Num -> Course Title holds for course and Course Num, Course Section -> Term, DoW, Time holds for Section.
Notice that this constraints (the FDs) are present in the real-world, and cannot be inferred just by looking at data. By looking at data we can say that an FD does not hold (if we have a pair of rows that violate the condition), but we can't be sure that it holds.

Lets try a few exercises to test our understanding of FDs. Given the following table:

Id	Name	Gender	Age
1	Orlando	Male	35
2	John	Male	35
3	Jane	Female	31
4	Jane	Female	30

Can **Id -> Name** hold ? Since there are no two rows with the same id, then yes (we can guess that this is true in the real world, with id being some form of identifier but it is just a guess until we verify its meaning in the real world)

Can **Age -> Gender** hold ? Well, we only have two rows with the same age (35), and they both have the same value for Gender, Male, so it can hold (here we would guess in the real world it doesn't hold, and at some point in time we could get a female who is 35)

Can **Gender -> Age** hold ? No, since for the two rows with female gender, they have a different value for age.

Can **Name,Age -> Id** hold ? yes, since there are no rows with the same values for both Name and Age

First Normal Form (1NF)

First normal form just requires that all attributes within a table are atomic, that is they are neither composite nor multivalued. We do not have any standard notation for tables which are NOT on 1NF, so all tables will be in 1NF by default.

Partial FDs and Second Normal Form (2NF)

We have said that the main issue with normalization is making sure that your table represents only one fact. Put in slightly more formal words, what you want is for every attribute to be dependent just on the keys of your table (remember, you may have more than one candidate key). One way in which you can break this rule is by having a **partial dependency**, in which some attributes are dependent on **just a part** of the primary key rather than the whole primary key (actually, since we may have more than one candidate key, it would be more accurate to say the determinant is a subset of a candidate key, but not a candidate key by itself). Notice this attributes would also be dependent on the PK, since the PK includes that part, but they are also dependent on just a piece of the PK.

For example, consider the table in the introductory example:

Course Num	**Course Title**	**Section Num**	**Term**	**DoW**	**Time**

With the following functional dependency:
CourseNum -> CourseTitle

This functional dependency is a **partial dependency**, since the determinant is a subset of the primary key (and not a key by itself). So, this table would be in 1NF but NOT in 2NF.
To transform this table into one in 2NF, we need to create a new table, comprised of all the fields in the offending functional dependency:

CourseNum	CourseTitle

Now, on this new table the dependency still holds, but it actually implies that CourseNum is the primary key of this new table (since it determines all other fields), so we should underline it

CourseNum	CourseTitle

Now, on the original table, we eliminate the fields determined by the offending FD (CourseTitle in this case), and add a foreign key relationship pointing to the new table, to be able to recover this information. The resulting relational schema diagram looks like this:

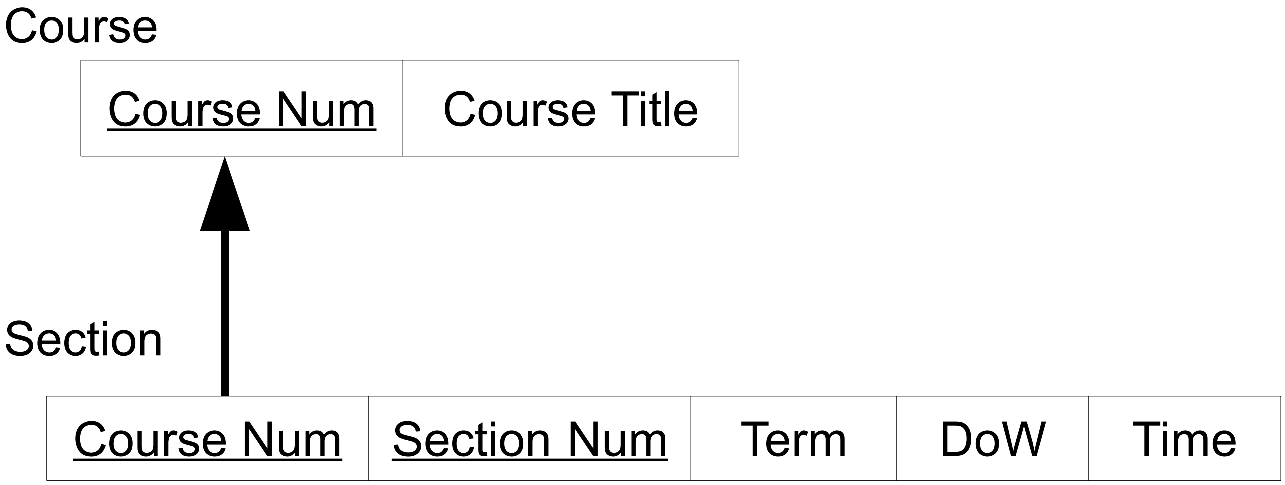


Figure 2: Converting to 2NF (repeat of Figure 1)

And this schema is in Second Normal Form (2NF). Notice the same functional dependencies still hold, it is just that the FD is now a full dependency on the new table.

Transitive Dependencies and Third Normal Form (3NF)

Another way in which we could have fields depending on something other than a candidate key is if we have **transitive dependencies**, that is, dependencies in which the determinant is neither a candidate key nor a part of any candidate key. We can have a table that is in 2NF but still has transitive dependencies. For example, assume we have a table like this:

StudentId	Name	MajorCode	MajorName

With the following additional functional dependency:

MajorCode -> MajorName

Now, this table would be in 2NF, since it doesn't have any partial dependencies; however, it would still lead to anomalies.

Since the determinant (MajorCode) is neither a candidate key nor a piece of it, we call the dependency a **transitive dependency**. To eliminate it, we again create a new table, containing all fields on this dependency:

MajorCode	MajorName

And again, the dependency implies that the determinant is the primary key for this new table:

MajorCode	MajorName

And on the original table, we eliminate the fields determined on the offending FD (MajorName in this case), and add a foreign key constraint from this original table to the new table, resulting in the following relational schema:

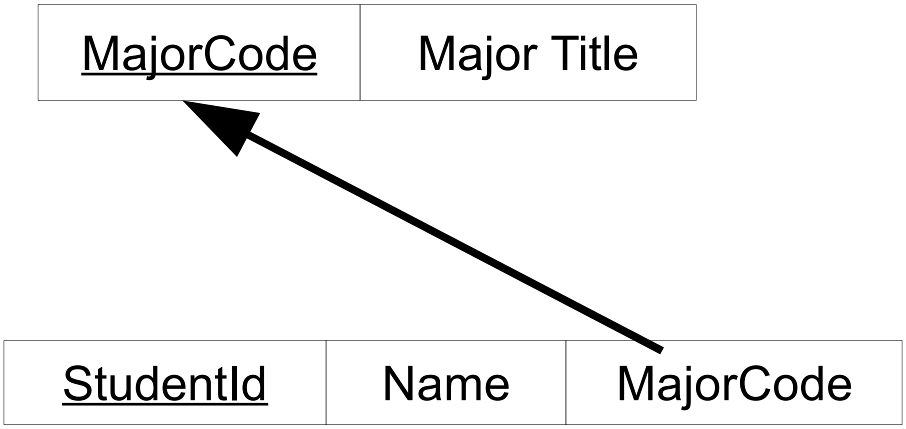


Figure 3: Relational schema for Students and Majors

Determining the Normal form of a Schema

Let's sumarize our definitions:

- A **partial** functional dependency is one in which the determinant is a proper subset of the primary key
- A **transitive** functional dependency is one in which the determinant is not a subset of the primary key
- A **full** functional dependency is one in which the determinant is the primary key

A relation is in:

- 1NF (First normal form) if it has any partial dependencies.
- 2NF (Second normal form) if it has no partial dependencies (normally, if it also has no transitiive dependencies we say it's in 3NF, which implies 2NF)
- 3NF (Third normal form) if it has no partial dependencies and no transitive dependencies.

And a relational database schema is in the lowest for of any of its relations; so it is in 1NF if any of its relations is in 1NF, 2NF if it has no relations in 1NF but some in 2NF and it is in 3NF if all its relations are in 3NF.

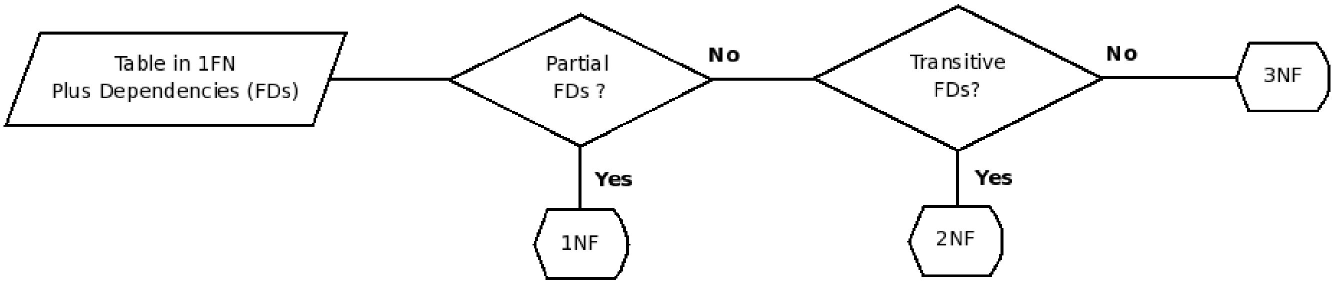


Figure 4: Flow diagram for determining the normal form a relational schema is in

Multiple candidate keys

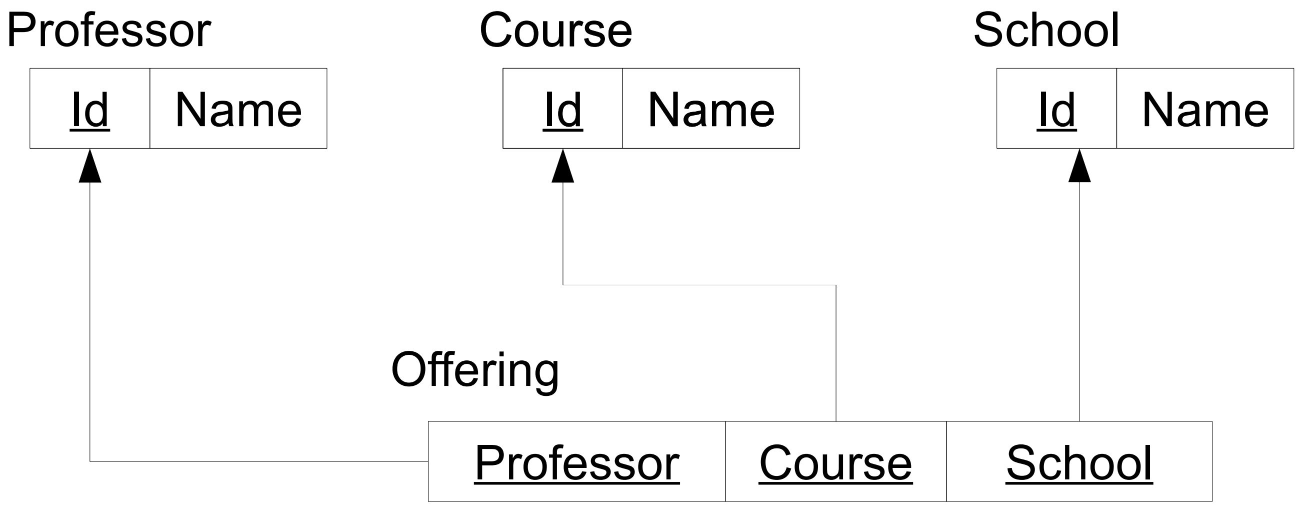
The above definitions deal only with the primary key. In many cases we have several **candidate keys** for a relation, and so the definitions need to be adjusted as follows:

- A **partial** functional dependency is one in which the determinant is a proper subset of a candidate key
- A **transitive** functional dependency is one in which the determinant is not a subset of the any candidate key
- A **full** functional dependency is one in which the determinant is a candidate key

Remember that we have the following definitions for keys:

- A **superkey** is any set of attributes that is guaranteed to be unique in a relation.
- A **candidate key** is a minimal superkey; that is a superkey that if we take any attribute out of it it is no longer a superkey
- The **primary key** of a relation is one of the candidate keys that is chosen to be used as foreign key in other relations.

Notice that every relation should have a primary key, and that the set of all attributes in a relation is always a superkey, since the relation is a set.



5. Examples and Exercises

1. We have a relation with fields A,B,C,D,E,F and its primary key is {A,B,C}; mark whether the following dependencies would be *partial*, *transitive*, or *full (OK)*.

Dependency	Status (partial, transitive, full)
A -> F	Partial (A is part of the PK, but not the full PK)
A,B -> E,F	Partial (A is part of the PK, but not the full PK)
E -> F	Transitive (E is not part of the PK)
A,B,C->E	Full (A,B,C is the PK)

2. We have a relation with fields A,B,C,D,E,F and its primary key is {D,E}; mark whether the following dependencies would be *partial*, *transitive*, or *full (OK)*.

Dependency	Status (partial, transitive, full)
A -> F	
E-> B	
E,F -> B	

3. For the following relation, and functional dependencies, indicate the normal form it is in (3 points). If it is not in 3NF, decompose it into 3NF relations.

Team

SportId	SportName	SchoolId	SchoolName	TeamName

Other Dependencies:

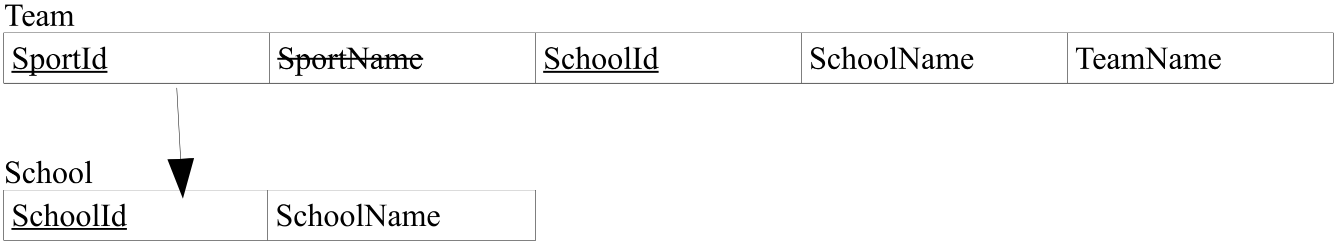
a. SportId -> SportName
b. SchoolId -> SchoolName

We can see that both of this dependencies are partial, because their **determinants** (SportId and SchoolId respectively) are pieces of the primary key. So, since there are partial dependencies, the relational schema is NOT even in 2NF (second normal form), which means it is in 1NF (first normal form)

Now, to transform this schema to 3NF, we need to eliminate the offending dependencies. Lets start with the first one, SportId -> SportName. We need to:

1. Create a new table, containing both attributes, with the determinant (SportId) as the primary key (we can call this table School)
2. Eliminate the determined attributes (SportName) from the original table
3. Add a foreign key constraint from the original table to this new table

The resulting schema looks as follows:



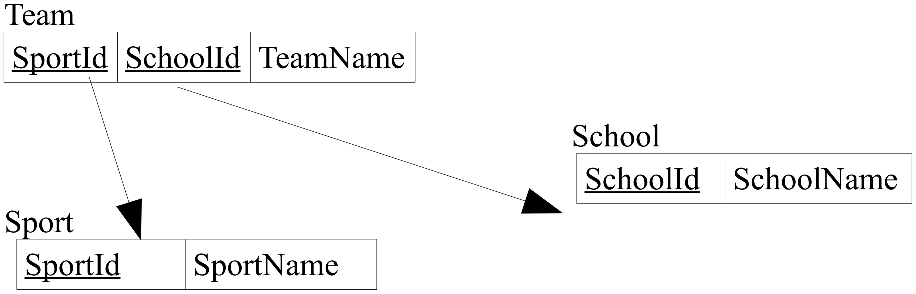
Team

SportId	~~SportName~~	SchoolId	SchoolName	TeamName

School

SchoolId	SchoolName

Now, this still leaves us with one partial dependency, SchoolId -> SchoolName; after a similar transformation (and completely deleting the unneede attributes from the original table, rather than just striking through), we end up with the following relational schema diagram.



Team

SportId	SchoolId	TeamName

School

SchoolId	SchoolName

Sport

SportId	SportName