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. Relational Schema Diagrams

It is oftentimes convenient to represent a relational database schema in a diagram, instead of listing sample values (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 (as in the example above) we can do:

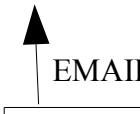
PERSON

Id	Name	Gender

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	Name	Gender



EMAILS

PersonId	Email

3. English, ER and the relational model

Of course, we can express the same knowledge from a database using natural language

4. Formalizing the model

We define a **tuple** as a mapping from names to values (a mapping is a partial function; you can also think of 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)* which is a mapping from names to domains (again, you can also think of it as 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*), all of them of the same type and compatible with the schema (basically 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.
- 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

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.

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.