Tips and Tricks for Project Task 2

Data Storage Paradigms, IV1351

1 Cardinality of a Relation

Here's **an example that illustrates exactly what the cardinality of a relation means**. Consider Figure 1, that shows the relation between the tables `account_holder` and `account` in a bank database. The cardinality on one side tells how many instances on that side one single instance on the other side can be related to. This means that one specific account must be related to exactly one holder, since the cardinality on the holder side is exactly one. One specific holder, on the other hand, can have any number of accounts, including zero, since the cardinality on the account side is minimum zero and without upper limit.

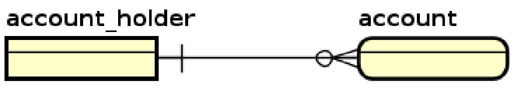


Figure 1: One holder can have any number, including zero, accounts. One account must have exactly one holder. Attributes have been omitted since they are not relevant to this example, but shall of course be included in a real database design.

2 Type of Surrogate Primary Key

It's a good practice to **use** `INT` **as the type for surrogate primary keys, and to also specify** `GENERATED ALWAYS AS IDENTITY`. With this declaration, the values of the primary key column will be given generated values, which are guaranteed to be unique, instead of having to be assigned manually. This declaration also removes the risk of accidentally manually inserting a non-unique value for the primary key, since `INT GENERATED ALWAYS AS IDENTITY` prohibits assigning a value manually. Finally, it's good to know that `GENERATED ALWAYS AS IDENTITY` is standard Sql, not any specific DBMS extension.

3 Avoid Duplicated Information

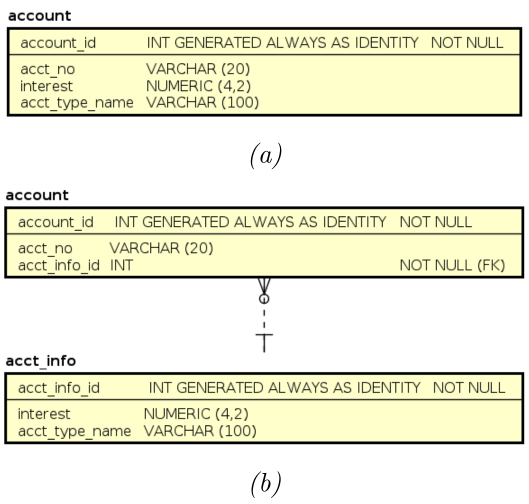

account
account_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
acct_no VARCHAR (20)
interest NUMERIC (4,2)
acct_type_name VARCHAR (100)
(a)
account
account_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
acct_no VARCHAR (20)
acct_info_id INT NOT NULL (FK)
acct_info
acct_info_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
interest NUMERIC (4,2)
acct_type_name VARCHAR (100)
(b)

Figure 2: The interest rate and account type name are duplicated on all rows of the account tables in (a), but stored only once in (b).

Duplicated information means the same information is stored in more than one place, which forces us to make sure the same value is always stored in all those places. As an example, let's consider bank accounts. Each account is of a specific type, let's say the bank has only two types, *savings account* and *salary account*. The interest is the same for all accounts of the same type. This information could be stored as in Figure 2a, but that would mean the same interest rate and account type name is stored on all rows describing the same type of account. In fact, no matter how many accounts there are in the bank, there are only two possible values of `account_name_type`, that is *savings account* and *salary account*. Likewise, there are only two possible interest rates. Such a solution creates severe problems. First, there's the risk that the wrong value is accidentally inserted when a new row is created. Second, if account name or interest is changed, it must be changed on all rows describing that type of account, which is both complicated and brings the risk of missing to update some row. Third, if no customer holds an account of a particular type, all information about that account type is lost. Fourth, the bank can't introduce a new account type without already having a customer that has opened an account of the new type. To solve these problems, the **duplicated data must be extracted to a separate table, which has a relation with the table that contained the duplicated data**. This is illustrated in Figure 2b.

4 How to Name Surrogate Primary Keys

It's best to establish a convention for the names of surrogate primary keys, and to always follow that convention. That minimizes the risk for confusion, since it becomes clear which columns are surrogate primary keys. There's no universally accepted best naming convention, but **it's common to call surrogate keys** `<table name>_id`, which means for example that the surrogate key of a table `account_info` will be called `account_info_id`. This has the advantage that the surrogate PK will have the same name as the FK of a table with a relation to the table with the surrogate PK. That allows natural joins (joins based on column name), which simplifies SQL queries. Exactly this naming is found in Figure 2b, where natural join can be used for the `account` and acct_info tables, for example using the query SELECT * FROM account NATURAL JOIN acct_info. The downside of the naming convention described here is that the table name is duplicated, since it's present in both table name and PK column name. Including the table name in a column name is generally a bad idea, because of this duplication, but for the purpose of enabling natural joins it's often used anyway in this particular situation. The main alternative, if we don't want a table name in a column name, is to call the surrogate PK just id, or perhaps key.

5 Primary Key of Table Representing Multivalued Attribute

This section is a warning of a possible mistake concerning the PK of a table representing a multivalued attribute. Let's for instance consider a table person, which has a multivalued attribute phone, meaning that one person can have multiple phone numbers. As usual, we create a separate table for the multivalued attribute, let's call it phone, and this table has a FK referencing the PK of the person table. The database now looks like Figure 3a.

This solution might be fine, but perhaps we didn't understand that now it's impossible for two persons to share the same phone number, since in that case there would be two rows with the same PK. Two persons can however share the same phone number if they have a land line. If sharing phone number shall be possible is up to the customer to decide, what matters to us is that we understand the consequences of our solution. **If we want to allow more than one row of a table to have the same value for a multivalued attribute, we have to include also the FK, in this case** person_id, **in the PK**, as is done in Figure 3b.

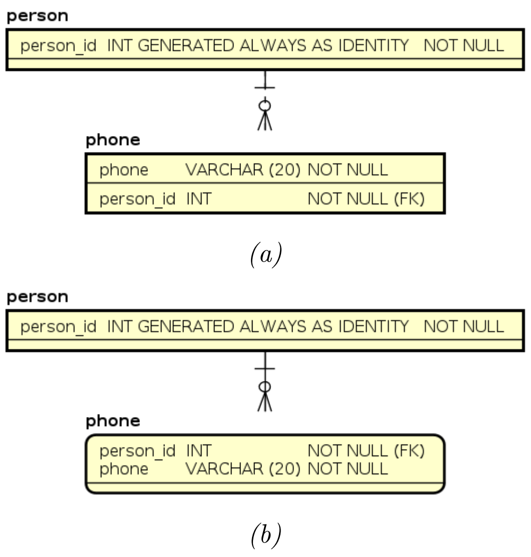

person
person_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
phone
phone VARCHAR (20) NOT NULL
person_id INT NOT NULL (FK)
(a)
person
person_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
phone
person_id INT NOT NULL (FK)
phone VARCHAR (20) NOT NULL
(b)

Figure 3: The database in (a) doesn't allow two (or more) persons to share the same phone number, while the solution in (b) does allow that.

Before leaving the subject of PK for a multivalued attribute, it might be worth also warning for the solution in Figure 4, which is completely wrong. Since the PK must be unique, there can only be one row in phone for each person, which means the attribute is in fact not multivalued at all.

6 Be Aware of Repeated Multivalued Attributes

Now let's look at the difference between a multivalued attribute and a many-to-many-relation. When to use which of the two? In the database depicted in Figure 3b above, we

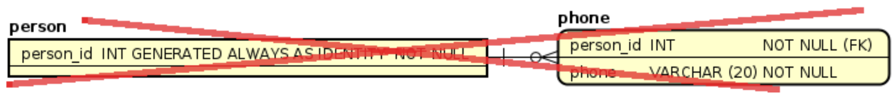
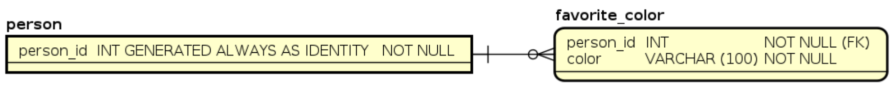
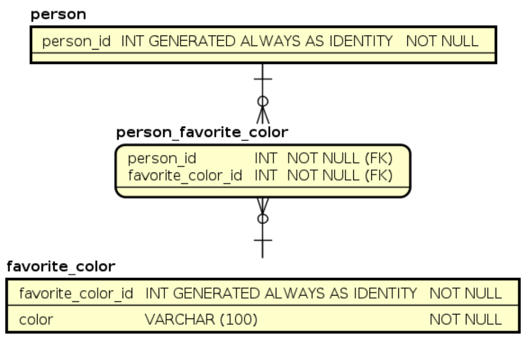

person
person_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
phone
person_id INT NOT NULL (FK)
phone VARCHAR (20) NOT NULL

Figure 4: It's ***wrong*** *to create a multivalued attribute like this, since the PK will have the same value for all values of the multivalued attribute belonging to the same row.*

actually have a kind of many-to-many situation, since one phone number can be used by more than one person, and one person can have more than one phone number. Why is it then modelled as a multivalued attribute, and not as a many-to-many relation with a cross-reference table? One necessary condition for using a multivalued attribute is that the attribute end is really just a value, if `phone` is replaced by something more complex, maybe for example `car`, which in its turn is a table with its own columns, then we must use a many-to-many relation. The question is if there are situations where a many-to-many relation is better even if the attribute end is just a value? The answer is 'yes', since **there will be duplicated information if many rows have the same value of the multivalued attribute**, similar to the case described in section 3. Consider the database in Figure 5a, storing persons and their favorite colors. With this solution, each color is stored once for each person having that favorite color.


person
person_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
favorite_color
person_id INT NOT NULL (FK)
color VARCHAR (100) NOT NULL

(a)


person
person_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
person_favorite_color
person_id INT NOT NULL (FK)
favorite_color_id INT NOT NULL (FK)
favorite_color
favorite_color_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
color VARCHAR (100) NOT NULL

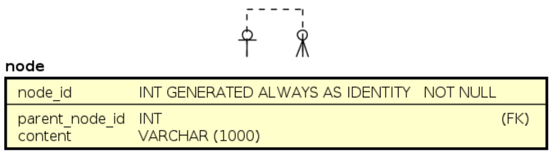
(b)

Figure 5: In (a), the same color will be stored once for each person having that favorite color. In (b), each color is only stored once.

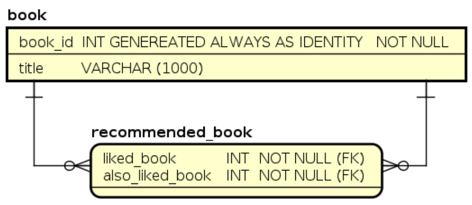
If we instead replace the multivalued attribute with a many-to-many relation, as in Figure 5b, each color will be stored only once, no matter how many persons have that favorite color. When to use which of these two alternatives is an interesting question. On one hand, the duplicated information in Figure 5a wastes storage space, and there's also the risk that the same color is stored different ways, for example using upper and lower case differently. On the other hand, we shall not be overambitious with normalization.

It's not necessarily a big problem if the same value appears in more than one place. As is the case in many situations, there's also here no exact rule defining when which of the two options is best. What matters most is that we're aware of the problem.

7 Relation to Self



(a)


book
book_id INT GENEREATED ALWAYS AS IDENTITY NOT NULL
title VARCHAR (1000)
recommended_book
liked_book INT NOT NULL (FK)
also_liked_book INT NOT NULL (FK)

(b)

Figure 6: Relations to self, one-to-many in (a), and many-to-many in (b).

Sometimes it's necessary to have a relation between rows of the same table. That might seem confusing, but in reality **there's nothing special at all with such a relation to self. It's handled exactly the same way as a relation to another table**. First, let's consider one-to-many (or one-to-one) relations. As an example, we can take a table containing nodes in a tree graph, which can be found in Figure 6a. Here, any number of nodes can have a relation to the same parent node.

Next, we'll look at the many-to-many relation to self depicted in Figure 6b. This database contains book recommendations, based on which books a reader likes. The cross-reference table `recommended-book` tells that a reader who liked `liked-book` might also like `also-liked-book`. Note that the cross-reference table is constructed exactly as if when used for many-to-many relations between different tables. What can be a little bit special in the case of relations to self, is exactly which rows to enter in `recommended-book`, since it's depends on whether recommendations are considered commutative and transitive. That is, will a reader who's recommended book two because of liking book one, also always be recommended book one because of liking book two? And if a reader liking book one is recommended book two, and a reader liking book two is recommended book three, will a reader liking book one be recommended book three?

8 Fewer and Bigger or More and Smaller Tables?

It's a frequently recurring decision whether to favor fewer and bigger, or more and smaller, tables. An example is a database containing person data. The persons may have a lot of attributes, including the home address. This address, in turn, has different parts, like street, zip and city. The question is then if these address parts shall be columns in a `person` table, or if a separate `address` table shall be created.

The main advantage of fewer and bigger tables is that SQL queries become easier to write, and faster to execute, since fewer `JOIN` operations are required. This is an important advantage, which often leads to favoring fewer and bigger tables. That's fine, but it's important to also be aware of the counterarguments. First, the problem with `JOIN`s can be to some extent remedied with views, and second, there are also disadvantages with fewer and bigger tables.

One disadvantage of bigger tables is duplicated information. In the `person` database example mentioned above, the same address will be repeated on all rows representing persons living at the same address. Another disadvantage is that also column names might be duplicated. Maybe not only persons, but also companies, have an address. Without an `address` table, there will in that case be `street`, `zip` and `city` columns in both the `person` table and the `company` table. Finally, it might be that more data is affected by locks when tables have more columns, since a lock on a row will affect all columns on that row.

9 Remember to Evaluate All Required Operations

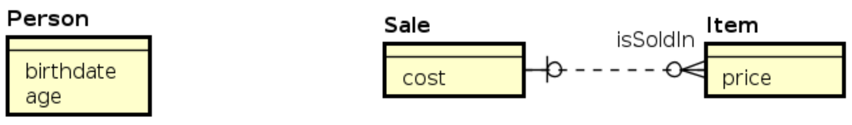
Before concluding that the database is complete, **make sure that it's possible to perform all operations that can be conceived**. This is definitely something that must be discussed with the customer, but since here there's no customer, we instead have to carefully read the business description and try to understand which operations are required. Below are some examples that illustrate how the description of Soundgood can be used to understand which operations the music school must be able to perform on the database.

- The fact that instructors are *available to give individual lessons during specified time periods*, tells that the database must show when an instructor is available to teach a particular instrument, at a particular skill level.
- Since students are *charged monthly for all lessons taken during the previous month*, the database must contain information about who took which lesson when. The same goes for instructors, since they are paid per lesson given.
- It must be possible to see who's currently renting which instrument, since Soundgood most likely wants to know where their instruments are.
- Since *group lessons and ensembles are given at scheduled time slots*, there must be information about this schedule in the database. The schedule reasonably must tell what is given when and where.

These were just a few examples, to illustrate how we might think. It's necessary to go through the entire description of the Soundgood music school and try to understand what operations are required.

10 Be Careful With Derived Data

This has already been covered in project task one, in the document *Tips and Tricks for Project Task 1*. The same reasoning is repeated here, in order to have everything related to task two in this document. Derived data is data that can be calculated from other data in the data storage. An example is an entity `Person`, that has an attribute `birthdate` and another attribute `age`, Figure 7a. `Age` is a derived data since it can be calculated from the birthdate, instead of being stored in an attribute. A less obvious example is found in Figure 7b, which models a part of a grocery store application. There's an entity `Sale` representing one particular sale to one particular customer. The entity `Sale` has a one-to-many relation with an entity `Item`, which represents something bought in the sale. If `Item` has an attribute `price`, which tells how much that item costs, and `Sale` has an attribute `cost`, which tells the total cost of the sale, then `cost` is derived, since it can be calculated from the prices of all bought items.



(a) *(b)*

Figure 7: Two examples of derived attributes. Age is derived in (a), and cost is derived in (b).

The disadvantage of derived data is that writing to the data storage is slower and more complicated, since more data must be written and more entities might be updated than if there wasn't any derived data. Also, derived data is against the spirit of relational data storage, which has data integrity as it's main goal. One important way to ensure data integrity is to never store duplicated data, which will in fact be done if we have derived data, since the derived data is a kind of duplication. The advantage of using derived data, on the other hand, is that reading is faster and maybe also easier when there is derived data. The bottom line is that we shall **not include derived data in a database unless we're very sure it's needed**. The reason is mainly the risk of loosing data integrity. Also, the read speed is very seldom a problem, and we must never optimize without being sure it's needed. That is, don't create derived data to improve read performance unless there's a measurement clearly showing that reads really are too slow. Also, the simplicity of reading data can be achieved in better ways than derived data, more on that when database queries are covered in project task three.

11 Actions Are Seldom Important, Don't Model a Program

For someone more used to programming than database development, there's the risk that program design sneaks into the database model. A clear sign of this mistake is that there are relations describing something that happens, instead of describing how data is related. This mistake can be illustrated by repeating the following example from the document *Tips and Tricks for Project Task 1*.

Imagine that we are modelling a database for a grocery store, and the description of the business says something like "the cashier scans the items the customer is buying". That might make us create a relation between the `Cashier` and `Item` tables, illustrating that *Cashier scans Item*, as depicted in Figure 8. Such a relation is, however, just plain

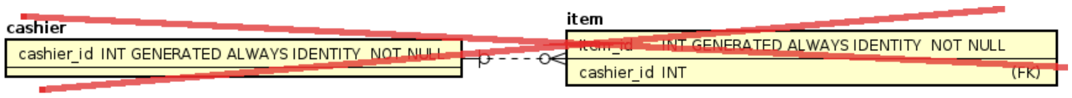

cashier
cashier_id INT GENERATED ALWAYS IDENTITY NOT NULL
item
item_id INT GENERATED ALWAYS IDENTITY NOT NULL
cashier_id INT (FK)

*Figure 8: It's **wrong** to create a relation with the sole purpose of showing that one entity does something with another entity. The purpose of relations is to show how data is related.*

wrong. The fact that cashiers scans items doesn't in any way relate data describing a cashier with data describing an item. To avoid such mistakes, **never see a relation as a call, and never create a relation with the intent of describing an action**.

12 Don't Create Data for Your Database Manually

There are many online services that can create Sql code for filling your database with data, for example `https://generatedata.com/`. It's much more efficient to use such a service than to create the data manually.

13 How to Know the FK Value When PK Is Generated

Making PKs `INT GENERATED ALWAYS AS IDENTITY`, as suggested in Section 2, means values of PKs are unknown in a script inserting data. This complicates assigning values to FKs. Say for example that we have a table `employee`, with a reference to a table `department`, since an employee works at a department, Figure 9. The script inserting data then contains `INSERT` statements creating departments, as in Listing 1. Note that no PK values are inserted, they are generated. The problem now is how to know which value to specify in the `department_id` FK when inserting employee data.

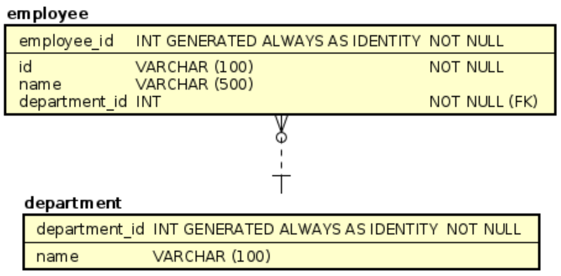

employee
employee_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
id VARCHAR (100) NOT NULL
name VARCHAR (500)
department_id INT NOT NULL (FK)
department
department_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
name VARCHAR (100)

Figure 9: Part of en employee database, where each employee works at one department.

```
INSERT INTO department (name) VALUES
    ('support'),
    ('sales'),
    ('develpment');
```

Listing 1: SQL statement for inserting departments

This can be solved as in Listing 2, where the `SELECT` clause in the `INSERT` statement, lines 3-4, retrieves the PK value for the searched department. That PK value is then inserted in the `department_id` FK column of the `employee` table.

```
1 INSERT INTO employee (id, name, department_id) VALUES
2     ('HYA984230948', 'Richard Jones',
3      (SELECT department_id FROM department
4       WHERE name = 'admin'));
```

Listing 2: SQL statement for inserting an employee working in the "admin" department

The nested `SELECT` in Figure 2 will be repeated each time an employee is inserted. If the `SELECT` is long, this will make the code hard to read. In that case, the repeated `SELECT` can be moved either to a view (Listing 3) or to a WITH clause (Listing 4). The main difference between a view and a with clause is that the view is stored in the schema and made available to all queries, while a `WITH` is only available in the query where it's written.

```
CREATE VIEW dep AS (
    SELECT department_id FROM department
    WHERE name = 'admin'
);

INSERT INTO employee (id, name, department_id) VALUES
    ('HYA984230948', 'Richard Jones',
     (SELECT department_id FROM dep));
```

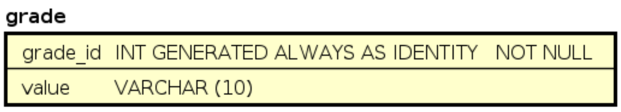
Listing 3: SQL statement for inserting an employee working in the "admin" department, using a view

```
WITH dep AS (
    SELECT department_id FROM department
    WHERE name = 'admin'
)
INSERT INTO employee (id, name, department_id) VALUES
    ('HYA984230948', 'Richard Jones',
     (SELECT department_id FROM dep));
```

Listing 4: SQL statement for inserting an employee working in the "admin" department, using a WITH clause

14 Enumerated Types

An enumerated type, often called *enum*, is a data type which can only have a set of named values. An example could be grades for a course. Maybe the allowed grades are just 'Pass' and 'Fail', which means the type `grade` can only have those two values. The by far most common way to handle this in a database is to create a separate table, sometimes called *lookup table*, which has the name of the enum, and contains one row for each possible value of that type. Considering the grades, this would look like Figure 10. Each column containing data of the `grade` type would then hold a FK indicating the row in the `grade` table that has the desired value.



grade

grade_id	INT GENERATED ALWAYS AS IDENTITY	NOT NULL
value	VARCHAR (10)	

Figure 10: The `grade` table represents an enumerated type called `grade`, and stores it's possible values.

Another, quite common, way to represent an enum in a database is to create it in the database schema, which for the `grade` type would be done with the statement in Listing 5. When the `grade` type has been specified like this, it can be used to specify column types just the same way as all other existing types are used.

```sql
CREATE TYPE grade AS ENUM ('Pass', 'Fail');
```

Listing 5: SQL statement creating en enum called `grade`, with the values `Pass` and `Fail`.

There are, however, many downsides of creating an enum as in Listing 5. One common argument against is that it's not standard Sql (the listing is for postgres). Each database vendor has their own implementation, with different syntax, and many don't have such a construct at all. Other arguments are that it's hard to change the values, if needed, that it's hard to list all possible values, and that it's not possible to add data related to a value, should the need arise. A perhaps more philosophical, but interesting argument against is that data is being treated like something else, not like data. The reasoning is that data in a relational database is what fills the tables, but creating an enum with Sql makes data (enum values) part of the schema instead.

A common argument in favor of creating an enum with Sql, as in Listing 5, is that queries become easier. The reason is that it removes the need to join a lookup table containing enum values, whenever a value of an enum is needed. That's however not completely true, since we can create a view combining the lookup table and the table which has a column of the enum type. Then that view can be used for all queries.

Before leaving this topic, it's worth warning for some other, completely useless, ways to implement enums in databases. One way is to use the type `VARCHAR` for the column containing the enum value, and just write the values in free text. This is begging for problems related to typos, or to different ways to represent a value, or mixing upper and lower case. As an example, 'Pass' and 'pass' would become two different values for the `grade` enum mentioned above. Another way of handling enums, that should also never be seen, is to replace them with numbers, for example using 1 for 'Pass' and 2 for 'Fail'. This is also begging for trouble, since we'll again and again get the burden of finding out which number means what, and translating them between number representation and text representation. Also, we're bound to sooner or later mix them and store the wrong number.