Logical Design

5 Logical design: Translating ER diagrams into SQL "`CREATE`" statements

In this section, we discuss how to translate the conceptual model of a database developed using Entity-Relationship modelling into a set of tables that should be implemented in a database system. This is termed the "logical design".

25. Schema notation. The standard notation used for describing tables is that of *schemas*. A schema for a table gives the table name followed by the attribute names in parentheses, with a few additional annotations. For example, the schema for the "`staff`" table above is written as:

```
staff(sid, title, firstname, lastname, email, office, phone)
```

By listing a schema for each table, we obtain a schema for an entire database.

26. Primary keys. In SQL databases, every table must have a *primary key*. The primary key is a set of attributes that *uniquely* identify a particular entity in the table. "Uniquely identifying" means that no two distinct records should have the same values for the attributes in the primary key.
For example, in the `staff` table, the `sid` field can be used as the primary key. We represent this by underlining the `sid` attribute.

```
staff(sid, title, firstname, lastname, email, office, phone)
```

If there are multiple attributes that form the primary key, you should underline all of them. It does not matter whether it is a single continuous underline or disjointed.

27. Allowing NULL values. Some fields in records can be NULL and others not. *The fields making up the primary key should never be NULL.* For other attributes, we indicate the possibility of NULLs by placing $^\circ$ superscript for all the attributes that we allow to be possibly NULL. All other attributes that do *not* have a $^\circ$ superscript are required to be non-NULL. Using these annotations, the schema for the "`staff`" table is written as follows:

```
staff(sid, title, firstname, lastname, email°, office°, phone°)
```

We are allowing the "`email`", "`office`" and "`phone`" attributes to be NULL but not "`title`", "`firstname`" and "`lastname`".

28. Attribute types In addition to the table name and attribute name, we must decide the types for each attribute. The possible types in SQL are as follows:

- `BOOLEAN` which has values `TRUE` and `FALSE`.
- `CHAR` which has values `'A'`, `'B'`, etc.
- `CHAR(`n`)` which has as values strings of length exactly n. If a shorter string is input then the system will pad it with blanks at the end.
- `TEXT` is the type of strings of arbitrary length (in practice, there is often an upper bound to the length as determined by the implementation of the DBMS but you are unlikely to come across this restriction).
- `VARCHAR(`n`)` is the type of strings of variable length but at most n. Longer strings will be rejected or truncated by the DBMS.
- `INT` (or `INTEGER`) is the type of integers in binary representation, just like Java's "`int`". In practice, integers have a maximum size, e.g. ± 2 Billion for 32-bit integers. *Only use `INT` when you know that you will stay within these bounds!*

- DECIMAL or NUMERIC is the type of numbers represented as strings of decimal digits. This will have some upper bound on the length of representations that the system is willing to tolerate. Although internally DECIMALs are stored just like strings, they are interpreted as numbers. Therefore any leading zeros are removed. On the other hand, trailing zeros are preserved because they could indicate a level of precision.

  There is also a type REAL which behaves like Java's "float" but this should only be used when small rounding errors are acceptable in the manipulation of values. The point here is that floating point numbers are represented in binary format and rounding in that format can be at variance with the rounding that one would expect from numbers in their decimal format. *Always use DECIMAL for amounts of money!*

- DECIMAL(n,m) or NUMERIC(n,m) which is the type of decimal integers where the overall number of digits does not exceed n, and exactly m digits appear after the decimal point.

- DATE which is the type of calendar dates. These can be input and output according to many regional conventions.

- TIME which has values in the formal 'HH:MM:SS'. Note that the numeric comparison operators $<$, etc., work for both DATE and TIME.

- SERIAL which provides a four-byte integer that is incremented automatically if a new tuple is inserted with DEFAULT as the corresponding value. This type can be used for generating unique id numbers. (This is a PostgreSQL-specific type. See below for the standard SQL mechanism for generating serial numbers.)

  You should note that the SERIAL type does not on its own entail that the corresponding field entries will be unique, as it is also possible to enter explicit values. In order to enforce uniqueness you must additionally declare the attribute to be a PRIMARY KEY or to be UNIQUE (see below).

- There are a number of additional types that you can learn about from the PostgreSQL documentation pages.

In case your data can not be represented as a value of one of these types you are probably dealing with a "complex attribute". These need to be broken down into attributes of primitive types. For example, to represent addresses as a field of type TEXT would be wrong in most cases because we would not be able to use the DBMS to retrieve individual aspects of an address, such as the city, or the country of residence.
An attribute, whose values are lists of variable length, should be represented as a weak entity; see the previous section.

29. CREATE TABLE statements. A table is created in SQL by writing a "CREATE TABLE" statement, which lists all the attributes, and attribute types and in addition, some *constraints* that the attributes must satisfy. For the time being, the only constraint that is marked is the primary key.

```
CREATE TABLE staff (sid       SERIAL PRIMARY KEY,
                    title     VARCHAR(6),
                    firstname VARCHAR(15),
                    lastname  VARCHAR(20),
                    email     VARCHAR(40),
                    office    INT,
                    phone     INT
                   );
```

30. Translating strong entities. Let's first consider a strong entity which is not involved in some generalisation hierarchy. In order to translate this into a table we proceed as follows:

a. Determine attributes.

b. Determine attribute types.

c. Determine a primary key. Usually, this requires the creation of an artificial attribute because real-world objects rarely carry unique identifiers with them. The primary key could be an integer but more typically is a string. This is because we want to avoid the DBMS removing leading zeros from our values.

d. Write out the SQL "CREATE" statement. For the

31. Declaring constraints. SQL "CREATE TABLE" statements allow us to declare not only types of the attributes but also additional constraints which the columns or tables must satisfy to maintain the integrity of the database. The "PRIMARY KEY" declaration above is an example of a constraint. Other common constraints are:

- "NOT NULL", which means that the column should not be null in any record, and
- "UNIQUE", which means that the values of the column should be unique across the table, i.e., distinct records in the table should have distinct values for this column.

The "PRIMARY KEY" declaration is in fact an abbreviation for the combination "UNIQUE NOT NULL".
Using these constraints the above "CREATE TABLE" statement can be refined as follows:

```
CREATE TABLE staff (sid       SERIAL PRIMARY KEY,
                    title     VARCHAR(6) NOT NULL,
                    firstname VARCHAR(15) NOT NULL,
                    lastname  VARCHAR(20) NOT NULL,
                    email     VARCHAR(40) UNIQUE,
                    office    INT,
                    phone     INT
                   );
```

This version of the statement imposes the "NOT NULL" constraint on "title" etc, and requires that the "email" field should be "UNIQUE" across the table.

32. Translating relationships. By default, a relationship is translated as a separate table. In order to represent the link to the entities involved we incorporate their primary keys as attributes. Thus they become "foreign keys" in the relationship table. Adding further attributes can be considered; often these refer to the time interval during which the relationship holds (or held) in the real world.
The foreign key attributes should be declared as such so that the system can keep an eye on the consistency of the stored information. In particular, it will return an error message when a record in an entity table is deleted that is still mentioned in some relationship table. The exact behaviour can be stipulated in the CREATE-statement:

- ON DELETE NO ACTION means that an entity record can not be deleted as long as some other table still refers to it. This is the default behaviour.
- ON DELETE CASCADE means that when a record of an entity is deleted, then so are all entries in tables that refer to it.

The "lecturing" table was created with the following command:

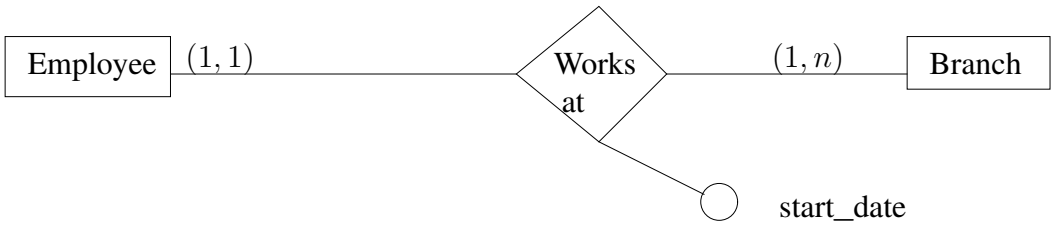
```
CREATE TABLE lecturing (cid     INT NOT NULL REFERENCES courses(cid),
                        sid     INT NOT NULL REFERENCES staff(sid),
                        year    INT NOT NULL,
                        numbers INT NOT NULL,
                        PRIMARY KEY (cid, sid, year)
                       );
```

(Note: We will see later that it was a bad idea to include the attribute "numbers" into this table.) The abbreviated schema notation for the above table is

```
lecturing(cid, sid, year, numbers)
```

The REFERENCES information is another example of constraints on the possible values in the database. Because it is about the entries in one table "referring" to entries in another table, it is called a **referential integrity** constraint.
The example shows two notations for declaring constraints. The NOT NULL and REFERENCES constraints are declared as constraints on individual columns. They are called "column constraints". The final PRIMARY KEY constraint is declared as a constraint on the entire table and is called a "table constraint". It is not possible to declare the "PRIMARY KEY" constraint as a column constraint in this instance because the primary key is made up of several columns.

33. Special cases regarding the translation of relationships. If the link between an entity E_1 and a relationship R is annotated with $(1,1)$ (meaning that each and every instance of the entity must take part in that relationship exactly once) then we can incorporate the relationship table into the table for E_1. The key of the other entity E_2 will then become an attribute in E_1 (i.e., a foreign key). Any additional attributes that we might want for R can also be incorporated into E_1.
As an example, consider the following fragment of an ER diagram:



This will result in two tables, one describing branches, the other describing employees. In the employees table we will have fields labelled "`branchID`" and "`start_date`". Since the design requires that every employee be based at a branch, we can further stipulate that the "`branchID`" field not be "`NULL`". This is expressed in SQL as follows:

```
CREATE TABLE staff (staffID    SERIAL PRIMARY KEY,
                    last_name  CHAR(20),
                    ...
                    branch     INT NOT NULL
                                   REFERENCES branch(branchID),
                    start_date DATE
                   );
```

If a link is annotated with $(0, 1)$ then we can still get rid of the relationship table but we must no longer stipulate that the foreign key field be "`NOT NULL`".

Finally, it may be the case that both links from the two entities to the relationship have maximal multiplicity 1. In this case the whole situation can be collapsed into a single table.

It is this coalescing of tables which is the reason why certain decision during the conceptual design phase do not always matter with regards to the number and structure of tables created in the end. Specifically, it will typically be irrelevant whether you model a relationship as a true ER relationship or whether you model it as a separate entity.

34. Sequence generators. To generate the unique serial numbers necessary for many applications the standard SQL (version 2003) provides a "sequence" feature. One can define a sequence for generating `sid`'s by:

```
CREATE SEQUENCE sid_seq;
```

The idea is that such a sequence generates an integer sequence 1, 2, 3, ... We can use the expression `nextval('sid_seq')` in SQL queries to get the *next* value in the sequence. In fact, we can also declare such an expression as the default value of the `sid` field:

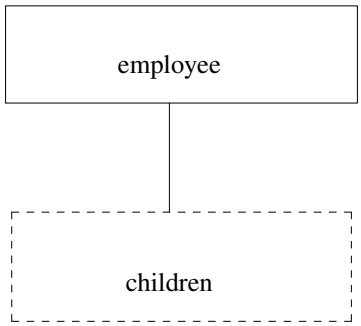
```
CREATE TABLE staff (sid    INTEGER DEFAULT nextval('sid_seq')
                           PRIMARY KEY,
                    title  VARCHAR(6) NOT NULL,
                    ...
                   );
```

The non-standard `SERIAL` data type of PostgreSQL is equivalent to a declaration of this form.

The expression `currval('sid_seq')` returns the "current value" of the sequence, i.e., the value last returned by `nextval`. This operator is useful to find out the id number of the last record inserted into a table, so that we can use it as a foreign key in other tables.

35. Translating weak entities. Remember that instances of a weak entity can not be disambiguated through their own attributes alone but that information about the parent entity is needed as well. From this it is clear that the table for the weak entity should have an additional column which contains the primary key value of the controlling instance. The primary key of the weak entity table will then consist of *two* columns: the primary key of the parent entity and an additional attribute to disambiguate entries that belong to one parent entry.

For an example, recall the situation we studied on the last section:

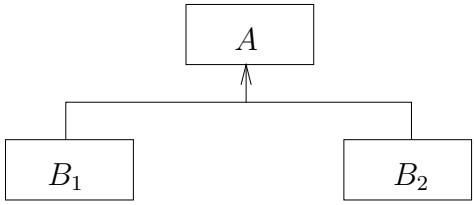


If the primary key of the "employee" table is called "employeeID", and we only want to record first name and date of birth for each individual child, then the "children" table should be created with the following SQL command:

```
CREATE TABLE children (first_name  CHAR(10) NOT NULL,
                       dob         DATE NOT NULL,
                       employeeID  INT NOT NULL
                                       REFERENCES employee(employeeID)
                                       ON DELETE CASCADE,
                       PRIMARY KEY (first_name, employeeID)
                      );
```

Note that we require "employeeID" to be present for each child and that we want information about children to be deleted when the parent leaves the company. Also note that the primary key consists of two attributes, as explained above, and that therefore we can not annotate an individual column declaration. Instead, we have made a separate PRIMARY KEY declaration *after* the last column has been declared.

36. Translating generalisation hierarchies. Consider the following (generic) situation with one superclass and two subclasses:

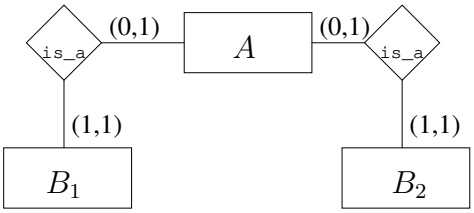


There are three possibilities for translating such a situation into tables:

a. Drop B_1 and B_2 and only represent A as a table.

b. Drop A and represent B_1 and B_2 as separate tables.

c. Represent each entity as a separate table. From a modelling point of view, this amounts to explicit "is_a" relationships between subclasses and superclasses (read this as "every instance of B_1 *is a* instance of A):



Because the multiplicities are as indicated, we will not get five tables from this model but three, as desired.

Let's discuss the pros and cons of each solution:

a. This is the commonest approach. The classification is indicated in the table for A by two "BOOLEAN" attributes which have value "TRUE" if the current instance belongs to the corresponding subclass. If the classification is exclusive (no overlap) then a single attribute will suffice.

This solution is problematic if instances of subclasses have wildly differing attribute sets because this will necessitate entering "NULL" for those attributes that do not apply for a given instance. It is also problematic if one subclass takes part in a relationship but others don't. Again, null values may then have to be used.

b. Dropping the superclass requires that the coverage by subclasses is total. Furthermore, we should also have that the subclasses do not overlap, in other words, that the classification is exclusive; otherwise we will have repeated entries which is always a bad idea in database design.

   From the querying point of view, this solution could also be problematic if it is common to ask questions that apply to all instances of A. In this case, the subclasses must each be queried independently and the results combined with "UNION". It is then important that no subclass is overlooked.

c. Having tables for both superclass and subclasses is the most general solution. It can always be adopted and does not rely on any special assumptions about the classification. However, it is also more expensive than the other solutions. Because common attributes are held at the A table and only additional attributes go into the B_i tables, more joins of tables will be involved in querying the system.

37. More on database constraints. We have already bumped into a number of ways of constraining what is accepted as a valid state of the database. Let's summarise this and complete the picture:

Field constraints The values that can legally appear under a single attribute can be constrained in three ways:

a. By assigning a type to each attribute. This information is in fact required when creating a table.

b. By requiring the presence of a value: "NOT NULL". This is optional; the default is that null values are allowed.

c. By formulating a specific condition on the possible values. The syntax for this is illustrated in the following example:

```
CREATE TABLE staff(salary DECIMAL(10,2) NOT NULL
                                        CHECK(salary >= 0),
                   ...
                  );
```

Record constraints These apply to rows of the table and typically relate the value in one column to that in another. Consequently, they are listed after each column has been declared. An example:

```
CREATE TABLE borrowings(rental_start DATE NOT NULL,
                        rental_end   DATE NOT NULL,
                        ...
                        CHECK(rental_start <= rental_end)
                       );
```

Table constraints These apply to the overall structure of the table, specifically to the question of whether certain values can be repeated. There are two forms:

a. A "PRIMARY KEY" declaration. If the primary key consists of a single column, its declaration can be made together with the declaration of that column. Otherwise, the declaration is placed at the end of the table definition. See this section for examples.

b. A "UNIQUE" declaration. Just like the previous declaration, this can either appear in a column declaration or at the end after all columns have been declared. It means that it is not allowed for two records to agree on all attributes mentioned in the "UNIQUE" declaration. We use this facility to make sure that secondary keys are indeed unique identifiers for records.

   Of course, a "PRIMARY KEY" declaration entails uniqueness.

Database constraints These span one or several tables in a given database. They come in several flavours:

a. *Referential integrity constraints* stipulate that the mention of a foreign key must relate to an actual entry in the table where this attribute is the primary key. The syntax is "REFERENCES" and we have seen examples above.

b. More generally, *enterprise constraints* allow the database administrator (or designer) to run a whole query whenever a certain action is performed on the data (such as an insertion of a new record). The syntax is illustrated in the following example:

```
CREATE ASSERTION conflict_free
       CHECK( 1 >= (SELECT MAX(lectures_per_hour)
                    FROM (SELECT COUNT(*) AS lectures_per_hour
                          FROM timetable
                          GROUP BY lecturer, timetable_slot)));
```

PostgreSQL does not support enterprise constraints.

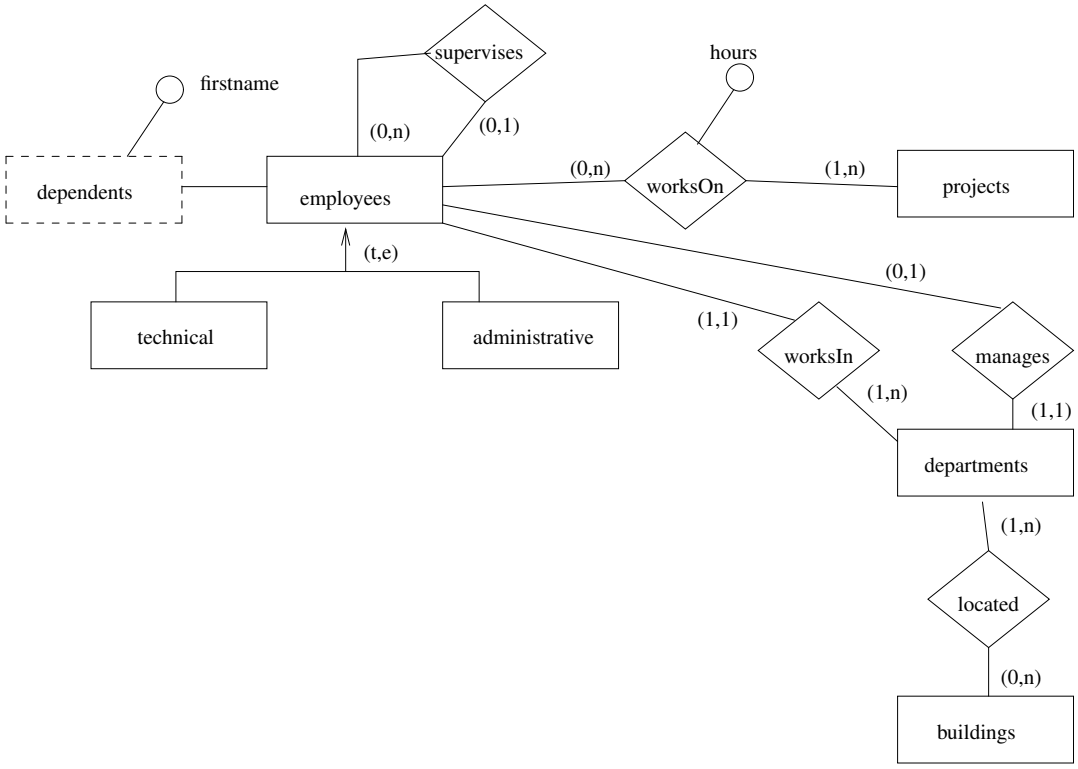
c. Another way to enforce enterprise constraints is to check data as and when it is entered into the database (or when records are updated or deleted) in the *application program* rather than relying on the DBMS to perform the check. Although this approach has the disadvantage that the task of keeping the data consistent is being split between two systems, it allows the designer to formulate conditions which require sophisticated processing and which are not easily expressed as SQL queries.

In general, you should keep in mind that constraints are a double-edged sword; on the one hand, they help us to keep the data in the database meaningful, but on the other hand, they may stop us from adapting the database design to a changing world.

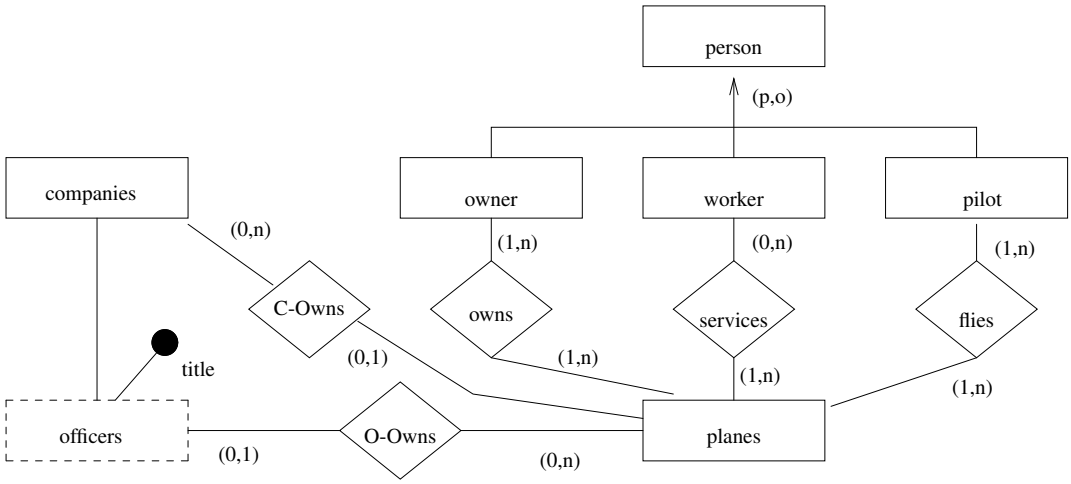
38. Miscellany: A bug in PostgreSQL. The database "`fundamentals`", which we have used for exercises and examples throughout the course, was created by Professor Jung and therefore he is the owner. Unfortunately, this does not stop anyone from creating tables in his database. Worse, we can not remove such tables because they will not belong to us.
Therefore, if you want to experiment with SQL `CREATE` statements, then **please do so in your own user space**. By default, `psql` will place you in your own database upon login anyway, so there is no excuse for inadvertently creating tables in "`fundamentals`".

39. Exercises The following entity-relationship diagrams are based on examples in the book by Batini, mentioned in the last section. For each of them perform the *logical design* step, that is, translate the diagram into SQL table declarations. Make sure that your SQL statements contain declarations of a primary key, and that all foreign keys are also declared. For the latter, specify a *delete*-policy.

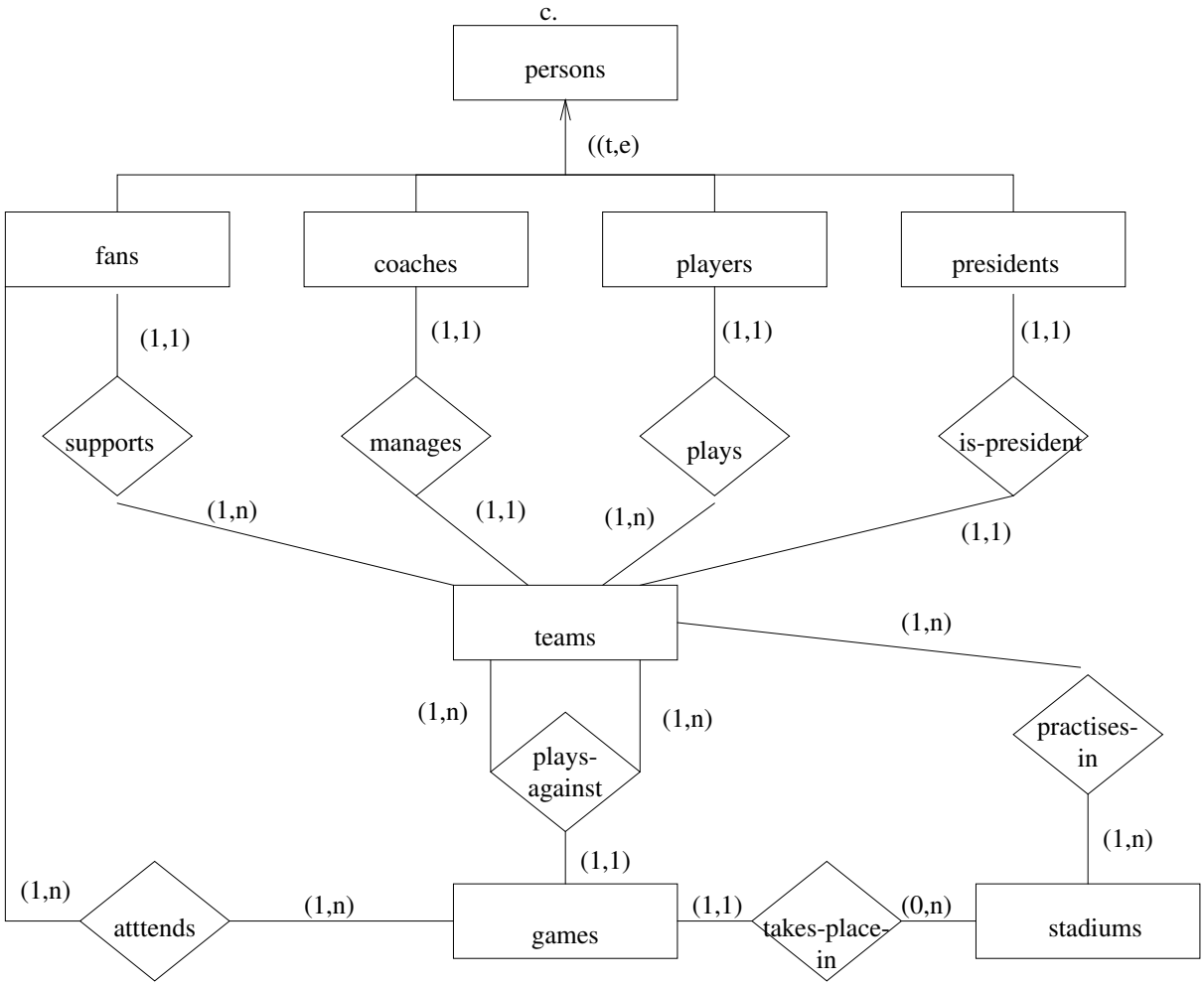
a.



b.


person
(p,o)
companies
owner
worker
pilot
(0,n)
(1,n)
(0,n)
(1,n)
C-Owns
owns
services
flies
(0,1)
(1,n)
(1,n)
(1,n)
title
officers
O-Owns
planes
(0,1)
(0,n)

c.


persons
((t,e)
fans
coaches
players
presidents
(1,1)
(1,1)
(1,1)
(1,1)
supports
manages
plays
is-president
(1,n)
(1,1)
(1,n)
(1,1)
teams
(1,n)
(1,n)
(1,n)
plays-against
practises-in
(1,1)
(1,n)
(1,n)
atttends
(1,n)
games
(1,1)
takes-place-in
(0,n)
stadiums