Relational Database Design and Relational Algebra

Jan Van den Bussche

Abstract

Notes for a part of the course on Data Management (Master Statistics and Data Science).

1 Introduction

Database management systems (DBMS) are complex software systems for the management of computerised data. DBMS are typically used by application programs (clients) that communicate with the DBMS (database server) in a query language, typically SQL. (We will learn the language SQL in this course.) A good DBMS allows many clients to use the system at the same time; this is a crucial property called *concurrency*. In this introductory course we will not have time to learn about concurrency.

A major distinction between DBMS and the standard file system provided by the computer's operating system is that a DBMS is based on a *logical data model*. Such a model allows the management of data on the logical level, without having to worry about how the data will be implemented in files. The most important logical data model is the *relational data model* and it is used by major DBMS (e.g., IBM DB2, Oracle, Microsoft SQL Server, MySQL, PostgreSQL).

You should know that also other logical data models are used, notably the RDF model used in the Semantic Web; the JSON model used in document-oriented databases such as MongoDB; and the "property graph model" used in graph databases such as Neo4J.

In this specific set of notes, we focus only on relational database design, and on relational algebra as the basis for SQL.

2 Preliminaries on graphs

Graphs are very important data structures in all sciences and computer science in particular. In short, a graph consists of *nodes* where some pairs of nodes are linked together by *edges*. Graphs are used whenever we need a data structure for representing connections between objects. Examples are:

- Road networks, where nodes are street crossings and edges are streets;
- Social networks, where nodes are persons and edges link together two persons who know each other;

- Infection networks, where nodes are organisms and an edge from x to y indicates that an infection has passed from x to y;
- Food webs, where nodes are species and an edge from x to y indicates that species x feeds on species y (e.g., rabbits feed on grass, specific bee species feed on specific flower species, specific carnivorous insect species feed on other specific insect species, such as ladybird beetles feeding on aphids).
- Large websites, where nodes are the different pages of the website and edges are links between webpages;
- The Internet, where nodes are computers and edges link together computers that are directly connected by a cable or wifi link;
- As we will see in this course, any kind of database can be thought of as a graph, where nodes are data entities and relationships, and edges indicate which entities participate in which relationships.

2.1 Mathematical definition of a graph[1]

Definition 1. A *graph* is a structure $G = (V, E, \mathit{tail}, \mathit{head})$ with the following components:

- V is a set; its elements are called the *nodes* of G.
- E is a set; its elements are called the *edges* of G. To avoid confusion between V and E, we always take them as disjoint sets (no common elements).
- $\mathit{tail} : E \to V$ and $\mathit{head} : E \to V$ are two total functions that indicate the tail and the head of every edge. For any edge $e \in E$ with $\mathit{tail}(e) = x$ and $\mathit{head}(e) = y$, we say that e is an edge from x to y.

Example 1. Consider the sets $V = \{v_1, v_2, v_3, v_4, v_5\}$ and $E = \{e_1, e_2, e_3, e_4\}$ and the following two functions from E to V:

e	$\mathit{tail}(e)$	$\mathit{head}(e)$
e_1	v_1	v_3
e_2	v_1	v_4
e_3	v_2	v_4
e_4	v_2	v_5

Thus we see that the tail of e_1 is v_1 and the head of e_1 is v_3, so that e_1 is an edge from v_1 to v_3.

The graph $G = (V, E, \mathit{tail}, \mathit{head})$ is depicted in Figure 1. As illustrated in the figure, graphs are usually depicted by placing the nodes in convenient positions and then indicating the edges by arrows. So, the names of the edges are not explicitly shown in the figure. As an exercise, you should write next to each arrow in the figure the name of the edge it represents. For example, the top leftmost arrow represents edge e_1 as it goes from v_1 to v_3.

[1] In the literature you will find many different mathematical definitions of graphs. The definition we use here is known as a "multigraph".

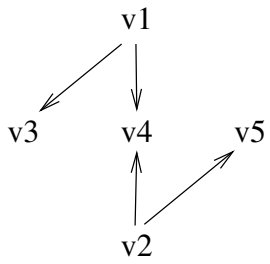

Figure 1: The graph from Example 1.

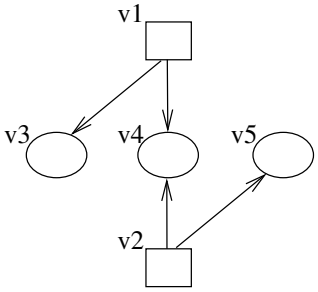

Figure 2: Many-sorted graph from Example 2.

2.2 Many-sorted graphs

It is often convenient to distinguish between different sorts of nodes in a graph. For example, in a food web, where nodes are species, we may consider various sorts of nodes, such as plants, insects, mammals, birds, fungi, etc. This leads us to the following extended definition:

Definition 2. A many-sorted graph is a structure $G = (V, E, S, \mathit{tail}, \mathit{head}, \mathit{sort})$ where

- $(V, E, \mathit{tail}, \mathit{head})$ is a graph as in Definition 1;
- S is a set; its elements are called the *sorts* of G;
- $\mathit{sort} : V \to S$ is a total mapping indicating the sort of each node of G.

Example 2. Let us use V, E, *head* and *tail* from Example 1 and introduce two sorts s_1 and s_2. So $S = \{s_1, s_2\}$. Consider the function *sort* defined by $\mathit{sort}(v_1) = \mathit{sort}(v_2) = s_1$ and $\mathit{sort}(v_3) = \mathit{sort}(v_4) = \mathit{sort}(v_5) = s_2$. Then the many-sorted graph $(V, E, S, \mathit{tail}, \mathit{head}, \mathit{sort})$ is depicted in Figure 2. As illustrated in the figure, sorts are often depicted by using special shapes for the nodes. In our example, we have used a rectangular shape for nodes of sort s_1 and an oval shape for nodes of sort s_2.

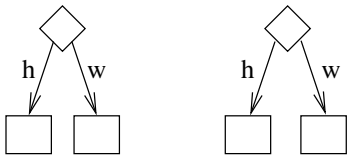

Figure 3: Example of a many-sorted, edge-labeled graph. Nodes of sort entity are shown as rectangles, and nodes of sort relationship are shown as diamonds. Edges are labeled with 'h' or 'w'.

2.3 Edge roles

For some applications it may also be convenient to be able to annotate the edges of a graph. Consider, for example, a graph representing people and their marital relationships. We have nodes of sort 'entity' that represent persons, and we also have nodes of sort 'relationship' that represent married couples. From each couple there are two edges to persons: one to the husband and one to the wife. To distinguish the husband from the wife (in the absence of any other properties of the nodes) we can label the edges with 'h' and 'w'. Such edge labels are called *roles*. A small example is shown in Figure 3. Note that in the figure, the names of the nodes have been omitted.

Formally, an edge-labeled many-sorted graph is defined as a structure

$$G = (V, E, S, R, tail, head, sort, role)$$

where

- $(V, E, S, tail, head, sort)$ is a many-sorted graph as defined before;
- R is a set elements called *roles*;
- $role : E \to R$ is a partial function giving labels to selected edges.

2.4 Node types

The final extension we make to our graph structures is that we can annotate not only edges, but also nodes. Such node annotations are called *types*.

Reconsider the example graph of Figure 3. That graph has nodes of sort entity and nodes of sort relationship. We could additionally make clear that nodes of sort entity represent persons by annotating them with the type 'Person'. Similarly we can annotate relationship nodes with the type 'Couple'.[2] This is shown in Figure 4.

The formal definition of a node-typed, edge-labeled, many-sorted graph follows the same pattern as before: it is as a structure

$$G = (V, E, S, R, T, tail, head, sort, role, type)$$

where

[2]In this example it may seem a bit redundant to distinguish nodes twice: by giving them a special sort ánd giving them a special type. But later we will see graphs where there are many nodes of sort entity, with many different types. Similarly for relationships.

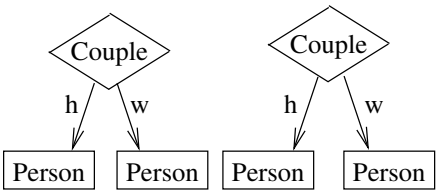

Figure 4: Example of a many-sorted, edge-labeled, node-typed graph.

- $(V, E, S, R, tail, head, sort, role)$ is an edge-labeled many-sorted graph as defined before;
- T is a set elements called *types*;
- $type : V \to T$ is a total function giving labels to selected nodes. Note that we define this function to be total, i.e., every node has a type.

From now on we will refer to node-typed, edge-labeled, many-sorted graphs simply as "graphs".

3 Introduction to database design

The steps in the design of a database are the following:

Requirements analysis: What data needs to be stored and managed? What use will be made of the database? What are the requirements of the system?

Conceptual design: Creation of a *conceptual schema* that defines the different types of entities and relationships that will consistute the information in the database. This schema is formulated in a *conceptual data model*. In this course we will use the Entity-Relationship (ER) model as the conceptual model, so the conceptual schema will be expressed as an *ER schema*. Another popular conceptual data model is UML class diagrams.

Logical design: The conceptual schema is a very high-level description of the information in the database. In particular, it is independent of the way the data will be structured inside the database, and how the data will be presented to the users. The definition of that structure and presentation is done by creating a *logical schema* from the conceptual schema. Since we are using the relational data model, this logical schema will be expressed as a *relational database schema*.

The conversion from the ER schema into the relational schema can be done automatically, but often the schema of the automated conversion will be too crude and will have to be refined manually. Still, the automatic conversion offers a good first approximation.

Physical design: The DBMS will automatically store data that is inserted in the database according to the logical schema. But a database administrator can instruct the system that the data must be stored in certain

special ways, so that the database becomes more efficient. Two aspects of physical design that we will briefly mention later in this course are the creation of *views* and the creation of *indexes*.

A natural question to ask is why conceptual design and logical design are two separate steps. Isn't it easier to go straight from requirements analysis to the creation of a relational database schema? The answer is that the conceptual schema is much easier to modify, so in the early stages of design, it is much easier to work with. Also, a conceptual schema is easier to understand for users that are not computing experts, but who may participate in the design of the database.

4 The entity-relationship model

In the entity-relationship (ER) model we conceptualize the contents of a database as a graph with three sorts of nodes: *entities*, *relationships*, and *attributes*. Edges can be from a relationship to an entity, meaning that the entity participates in the relationship. Some of these edges carry a role. Edges can also be from an entity to an attribute, or from a relationship to an attribute. Entities, relationships, and attributes have types.

4.1 Entities

For example, think of a university database. We can think of many entities "living" in the database: students, professors, courses, etc. We may store for each student some attributes such as name and birthdate. Similarly we may store attributes for professors such as name and salary. We can thus conceptualise the content of the database as a graph with entity nodes of types 'student' and 'professor' and attribute nodes of types 'name', 'birthdate' and 'salary'. Edges connect entities with their attributes. A small example with two students and two professors is shown in Figure 5. Such a graph representing a possible content of the database is called an *Entity-Relationship instance*, or ER-instance for short. The convention is that entity nodes are shown as rectangular shapes, and attribute nodes as oval shapes. Note also that edges go from entities to attributes, but in figures we normally do not draw the arrows and just represent the edges as lines.

4.1.1 Well-formedness requirements

An ER instance has to be *well-formed*. Thereto, it must satisfy a number of natural requirements:

Single attribute value constraint: Each entity node never has more than one attribute value of a given type. More precisely, it is illegal to have an entity node e and two distinct attribute nodes a_1 and a_2 such that a_1 and a_2 have the same type and there are edges from e to a_1 as well as from e to a_2.

Uniform attributes constraint: Entity nodes of the same type should have the same attribute types. More precisely, if e_1 and e_2 are entity nodes of

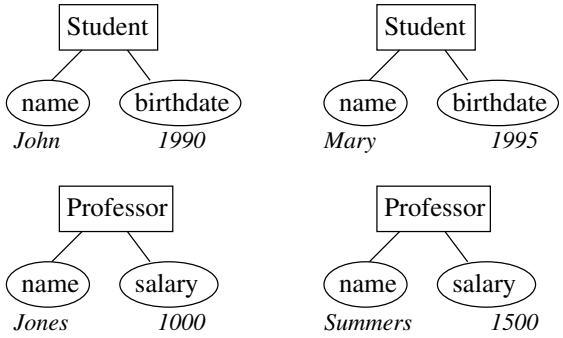

Figure 5: A very small ER instance consisting of two students and two professors.

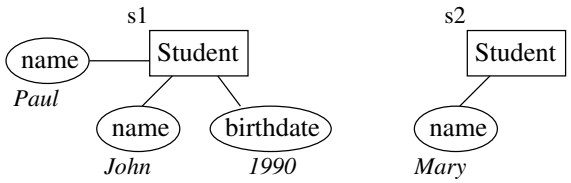

Figure 6: This graph is not a well-formed ER instance, for multiple reasons. First, the single attribute value constraint is violated by entity s_1 because it has two attributes of type 'name'. Moreover, the uniform attributes constraint is violated because s_1 has an attribute of type 'birthdate', but s_2, which is of the same type student as s_1, does not have an attribute of type 'birthdate'.

> the same type, and e_1 has an edge to some attribute node a_1, then also e_2 must have an edge to some attribute node a_2 of the same type as a_1.

These constraints are illustrated in Figure 6.

4.2 Relationships

A database containing only entities is not very interesting. Indeed, also in the real world, entities are related through relationships. Let us see some examples.

Example 3. In a database about students and courses we have entities of type *student* and *course*. Course entities have the title of the course as an attribute. To record which students take which courses, we introduce relationships of type *exam*. To indicate that student s takes course c, we add a relationship node r of type *exam* and link it by edges to s and c. We can also attach attributes to relationship nodes; for example, we may add the score of the exam as attribute. A simple ER instance illustrating this example is shown in Figure 7. The convention is that relationship nodes are shown as diamond shapes. In ER instances, edges always run from relationship nodes to entity nodes, but in figures the arrows of these edges are omitted as their orientation is understood.

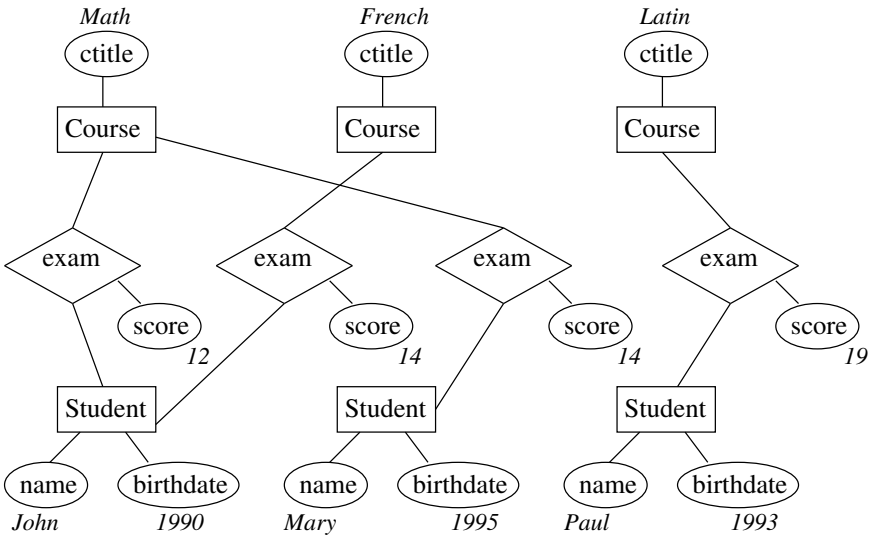

Figure 7: ER instance with relationship nodes.

Example 4. We could also have relationships of type *teaches* to relate professors to courses. Such relationships could have two attributes to indicate the room and the time of teaching.

Example 5. Relationships do not have to be binary, i.e., relating just two entities. For example, instead of attributes we could decide to create rooms as separate entities. Then the teaching relationship would be a ternary, relating courses, professors, and rooms.

Example 6. Relationships can also relate entities of the same type. For example, consider professors serving as a stand-in for other professors for certain courses. This is a ternary relationship type, relating professors to other professors and to courses. In such a case, we need to use *roles* to discriminate the different roles of the entities taking part in the relationship. In the stand-in example, we use roles *before* and *after* to distinguish the professor that is being replaced (before) from the professor that serves as a stand-in (after). These relationships could also have an attribute to record the year when the replacement occurred. A small ER instance illustrating this example is shown in Figure 8. Note the roles that annotate the edges from stand-in nodes to professor nodes.

The blank role When an edge from a relationship node to an entity node has no role, it is convenient to still think of it as carrying a role, namely, the *blank* role. The blank role is distinct from all other roles.

4.2.1 Well-formedness requirements, continued

As with entity nodes, ER instances should also satisfy some well-formedness requirements related to relationship nodes.

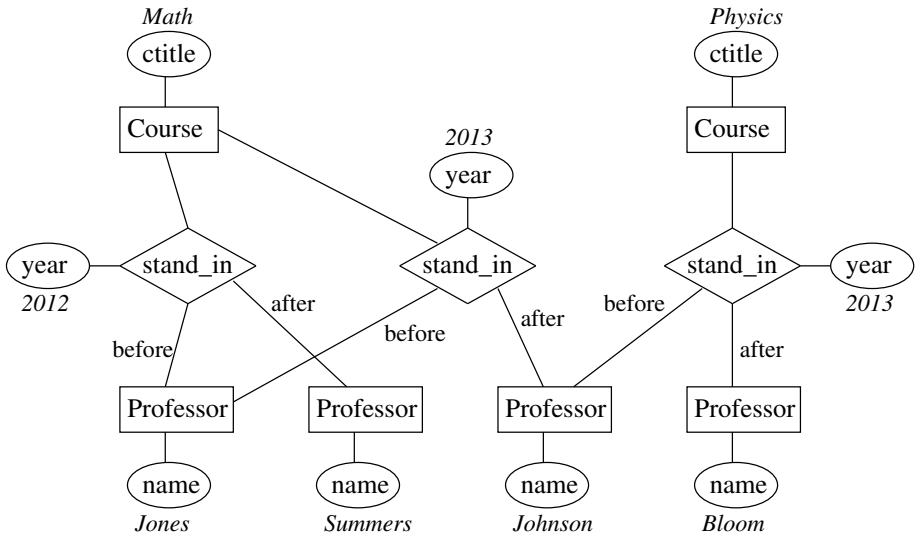

Figure 8: An ER instance with edge roles.

Single attribute value constraint: Must hold for relationship nodes in the same way as stated earlier for entity nodes.

Uniform attributes constraint: Idem.

Distinguishing roles constraint: When a relationship node has two edges to entity nodes of the same type, or even two edges to a same entity node, then these two edges must have distinct roles.

Uniform roles constraint: Relationship nodes of the same type must have similar edges to entities. More precisely, if r_1 and r_2 are two relationship nodes of the same type; e_1 is an entity node; l is a role (possibly the blank role); and there is an edge from r_1 to e_1 with role l, then also r_2 must have an edge with role l to some entity e_2 of the same type as e_1.

Unique relationship constraint: A relationship node represents a relationship between the participating entities in their respective roles as indicated by edges from the relationship nodes to two entity nodes. It is not allowed to have to distinct relationship nodes of the same type, representing exactly the same relationship, even if the two nodes would have different attributes.

More precisely, consider any relationship node r. Let $e_1, \ldots, e_n$ be the edges going out from r to entity nodes $v_1, \ldots, v_n$ respectively. And let $l_1, \ldots, l_n$, respectively be the roles on these edges (some of the l_i may be blank). Then we say that r represents the relationship $(l_1 : v_1, \ldots, l_n : v_n)$. Now it is illegal for there to exist another relationship node r', distinct from r but of the same type of r, so that r' also represents the relationship $(l_1 : v_1, \ldots, l_n : v_n)$.

The distinguishing roles constraint and the uniform roles constraint are illustrated in Figure 9. The unique relationship constraint is illustrated in Figure 10.

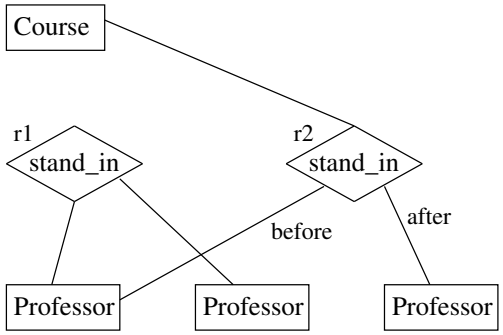

Figure 9: This graph is not a well-formed ER instance for at least two reasons. First, the distinguishing roles constraint is violated, since node r_1 has two edges to entities of the same type (Professor), yet the two edges do not have distinct roles. Also the uniform roles constraint is violated: r_2 has an edge with blank role to an entity of type Course, but r_1, which is of the same type as r_2 (stand_in), does not have a blank edge to an entity of type Course.

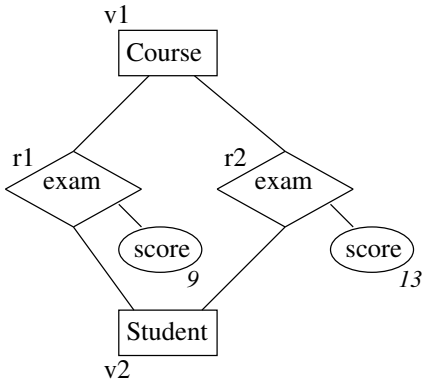

Figure 10: This graph is not a well-formed ER instance because it violates the unique relationship constraint. Relationship nodes r_1 and r_2 are of the same type (exam) and both represent the same relationship, namely, $(b : v_1, b : v_2)$ where by 'b' we mean the blank role. This is forbidden by the unique relationship constraint.

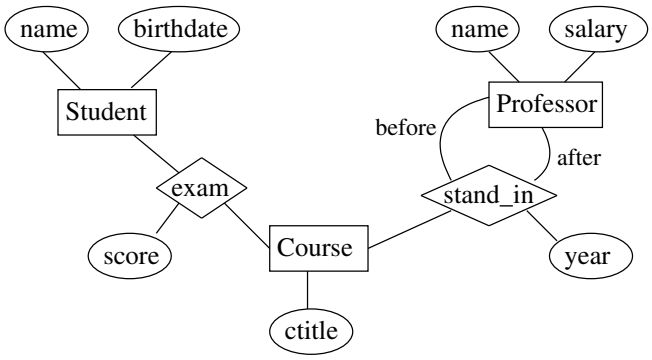

Figure 11: Example of an ER-schema.

Question: In Figure 10 we have seen that conceptualizing an exam as a relationship between students and courses, with the score as attribute, does not allow multiple exams to be taken by a same student for a same course, with different scores. How would you conceptualize database instances where such multiple exams are possible?

4.3 ER Schemas

An ER instance describes a snapshot of the contents of the database. When designing a new database, we must think of which possible instances we want to allow in our database. Thus, we must think of which entity types, which relationship types, and which attribute types we want in our database, and how these types relate to each other.

A summary of the allowed entity types, relationship types, attribute types, and their relations, is given in the form of an *Entity-Relationship schema*, of ER-schema for short. An ER-schema is a graph just like an ER-instance, but is not intended as an ER-instance itself: rather, it serves as a template that describes the structure that all ER-instances should conform to. Thus, an ER-schema has only one node per entity type, one node per relationship type, and one node per attribute type. The edges in the ER-schema show how all the different types are allowed to relate to each other in an instance. Like instances, ER-schemas must be well-formed; they must satisfy the distinguishing roles constraint. The other well-formedness constraints are trivial for schemas, since there is only one node for each type.[3]

Example 7. An example of an ER-schema is shown in Figure 11. The depiction of an ER-schema is also often called an "ER-diagram".

An ER-schema serves as a template, describing the node types and edges that are allowed in instances. Thus, an instance is allowed if it can be "matched" against the schema. In that case we say that the instance conforms to the schema. We will omit a formal definition of conformance, and instead give some examples.

[3]In figures of ER schemas, we will duplicate attribute nodes to avoid cluttering up the drawing. For example, in the ER schema shown in Figure 11, there is really only one attribute node of type 'name', but we have drawn it twice to make the drawing more clear.

Figure 12: The instance on the left does not conform to the schema on the right. Indeed, in the instance, the Student entity nas a birthday attribute, but in the schema, only a name attribute is declared for students.

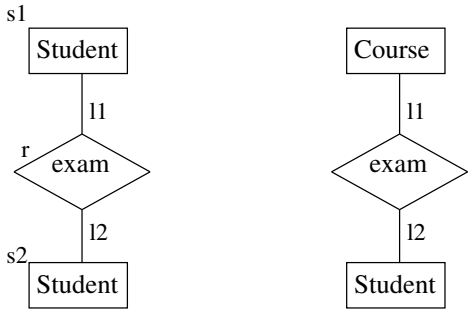

Figure 13: The instance on the left does not conform to the schema on the right. In the instance, the exam relationship relates two students (s_1 and s_2). However, the schema prescribes that exam relationships should relate students to courses.

Example 8. The ER-instance of Figure 5, and the ER-instance of Figure 7, both conform to the ER schema of Figure 11. The ER-instance of Figure 8 *almost* conforms to the ER schema, but not quite! Indeed, in the schema, the entity type Professor has two attributes name and salary, but in the instance, each Professor has only a name attribute but no salary attribute.

Example 9. Figures 12, 13 and 14 give simple examples of instances that do not conform to certain schemas.

4.4 ER-schemas with keys and functional edges

ER-schemas as defined so far still lack two important ingredients that are commonly used by database designers to further limit the allowed instances: *keys* and *functional edges*. We next introduce these two additions to an ER-schema.

4.4.1 Keys for entity types

Given an entity type in an ER-schema, we must designate a *key* for it. This key is simply a subset of the attributes of the entity type. We will indicate the key in an ER-diagram by underlining the attribute types that belong to the key.

In an ER-schema, we are expected to designate a key for each entity type of the schema. A well-formed ER-instance is said to *conform* to an ER-schema

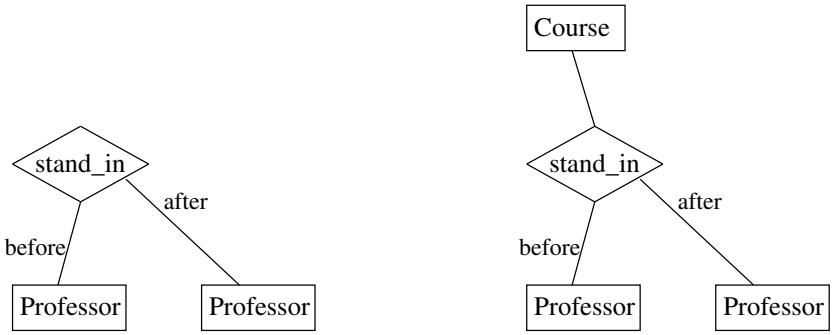

Figure 14: The instance on the left does not conform to the schema on the right. In the instance, the stand_in relationship relates two professors, but the schema prescribes that a stand_in relationship should relate two professors with a course.

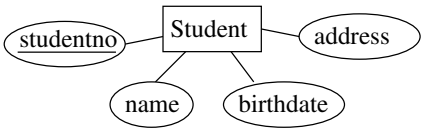

Figure 15: ER diagram with key attribute underlined.

with keys, if the instance conforms to the schema in the sense already seen previously, and, furthermore, the instance *satisfies all the keys*. By this we mean the following:

- Let E be an entity type in the schema, and let K be the set of attribute types that make up the key of E. Then in the instance there must not exist two distinct entity nodes v_1 and v_2 that are both of type E and that have exactly the same values for all attributes from K. In other words, any two distinct entities of type E must differ in the value of at least one attribute from the key.

Example 10. Consider an entity type Student with attribute types studentno, name, birthdate, and address. Suppose we designate the singleton {studentno} as key for type Student. In an ER diagram this would be depicted as shown in Figure 15. Then the instance shown in Figure 16 does not satisfy the key.

Example 11. A key may consist of multiple attributes. For example, consider an entity type Diary Entry with attribute types year, month, day, hour, minute, and text. The attribute text contains the text of the diary entry, and the year, month, day, hour and minute attributes together indicate precisely when the entry was made. Then it is reasonable to designate the set of attributes $\{\text{year}, \text{month}, \text{day}, \text{hour}, \text{minute}\}$ as the key for this entity type.

Remark For some entity types we may designate the set of *all* attributes of that type as the key for that entity type. It simply means that we do not allow

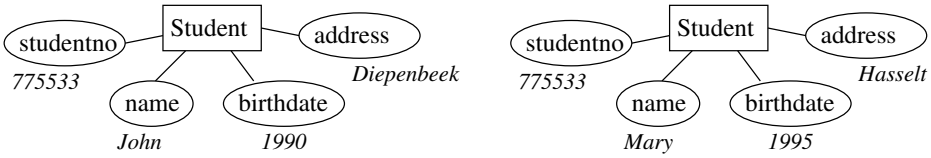

Figure 16: This ER-instance does not satisfy the key {studentno} for Student, because the instance contains two distinct entities of type Student that have the same value for attribute studentno.

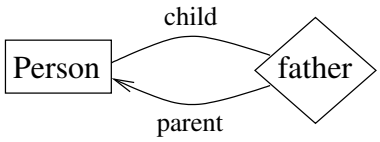

Figure 17: Note the arrowhead to indicate that the edge of the relationship type is functional.

two distinct entities with exactly the same values for all attributes, i.e., any two distinct entities must differ in at least one attribute value. This is a perfectly reasonable thing to specify, if no proper subset of attributes can be expected to act as a key.

4.4.2 Functional relationship edges

In an ER schema, we may specify for some of the edges of some of the relationship types that these edges are *functional*. We begin with an example.

Example 12. Consider an entity type Person and a relationship type 'father' between Person (with role parent) and Person (with role child). Clearly every person can have at most one father.[4] We say that *the edge with role 'parent' of relationship 'father' is functional*. In an ER diagram, this is indicated by placing an arrowhead at the head of a functional edge. This is illustrated in Figure 17.

Terminology: many-to-one and one-to-one A relationship type with two edges is called a *binary* relationship type. When one of the edges is functional but the other is not, such a relationship type is called "many to one". For example, the father relationship is many to one because every person has one father, but a father may be the parent of many children. When both edges of a binary relationship type are designated as functional, the relationship type is called "one to one". An example would be the relationship type "married" between Person with role husband and Person with role wife. (In most cultures,)

[4]Of course in the real world, every person has exactly one father, but in the database, it is impossible to store the father of every person in the database: we would then also have to store the father of each father, and so on, all the way back to Adam! So, in our database, we can store the father for some persons, but not for all, i.e., in the database, every person has at most one father.

both edges are functional, since every person is the husband of at most one person, and every person is the wife of at most one person.

Example 13. Functional edges are not restricted to binary relationship types. Recall the relationship type 'stand_in' from the schema of Figure 11. This relationship type is ternary (has arity three, i.e., has three edges). If we would designate the edge with role 'after' as functional, it would mean that we allow only one replacement for each professor–course combination.

We now formally define what it takes for an ER instance to *satisfy* a functional edge. Let H be an ER-schema. Let R be a relationship type in H. Let e be an edge out of R to entity type E, with role l. We number e, E and l as e_1, E_1 and l_1, and let all the other edges going out from R be $e_2, \ldots, e_n$, to entity types $E_2, \ldots, E_n$, with roles $l_2, \ldots, l_n$, respectively.

- An instance I conforming to H is said to *satisfy* the functionality of e if the following holds. Consider any combination of $n-1$ entity nodes $v_2, \ldots, v_n$ in I, of types $E_2, \ldots, E_n$ respectively. Then there can be at most one entity v_1 in I such that the relationship $(l_1 : v_1, l_2 : v_2, \ldots, l_n : v_n)$ is represented in I by some relationship node of type R.

Example 14. (Example 13 continued.) Let e be the edge with role 'after' for the stand_in relationship type. The instance we have already seen in Figure 8 does not satisfy the functionality of e. Indeed, consider the course Math and the professor Jones. This combination is involved in two different relationships: once with the professor Summers, and once with the professor Johnson.

4.5 Conclusion on database design

Based on the needs of the application, the database designer will design an ER-schema with keys indicated for each entity type, and some edges of some relationship types indicated as functional. The intention by doing that is to make precise that the intended database contents are ER-instances that must conform to the ER-schema and must satisfy all keys and all functional edges that are indicated.

In practice, large ER diagrams may consist of hundreds of entity types and relationship types, and are not always easy to manage or to understand. There are software tools available that allow the user to create and manage ER diagrams on the computer. Such tools are known as *CASE tools* (Computer-Aided Software Engineering). A quite famous such tool is called Rational Rose, currently marketed by IBM. Rational Rose uses UML rather than the ER model; UML is a very large conceptual modeling language that encompasses not just databases but the entire software design process. The database diagrams of UML (called class diagrams) are very similar to ER diagrams. A serious limitation of UML class diagrams, however, is that they allow only binary relationships.

The design of an ER schema is not an easy task. Entire books have been written on the art of conceptual modeling; all we can do in this limited space is give a few tips.

Use object types from the real world as entity types. When you have to design a database that automates the information of a certain enterprise, take a good look at the enterprise and see what are the important types of objects that play a role in the enterprise. This is a good start to determine what the entity types will be in your information system. Note that entity types can also represent purely administrative, or purely theoretical objects, e.g., an *invoice* can be a perfectly good entity type in a commercial information system, or a *disease* can be an entity type in a healthcare information system.

If the entity types in your schema represent something from the real world, it will also be much easier to determine the relevant relationship types for your system.

Use attributes only for single values; use relationships for everything else. Consider an entity type *article* for scientific articles. Such articles often have many authors. A bad design would be to put an upper limit on the number of authors, say ten, and provide ten attributes for *article*, say, author_1, author_2, ..., author_{10}. Much better in this case is to create a separate entity type *author* and a relationship type *author_of* between authors and articles. The authors are typically listed in some order: the position number of an author in the author list of an article could then be recorded as an attribute of *author_of*. Since an article has only one title, it is appropriate to use title as an attribute for *article*.

Don't be afraid to use non-binary relationships. A binary relationship relates two entities. Often higher arities are appropriate. Consider again the relationship type *exam* that relates students and courses, with an attribute score. If a student fails for an exam, i.e., if his score is too low, he needs to take it again. But this is not supported by our relationship type: for each student and each course there can be at most one relationship of type *exam* relating these two entities. To support taking multiple exams, we can create an entity *semester* with attributes year and period, for example. Then, *exam* can become a ternary relationship relating students, courses, and semesters. We keep the attribute for the score. In this way, a student can take an exam for a course every semester, and we record all the scores of all the trials.

Table 1: Example of a relation instance over the relation scheme {number, name, birthdate, address, year}.

number	name	birthdate	address	year
123	john	1980	diepenbeek	2006
456	mary	1985	hasselt	2007
789	mary	1985	hasselt	2007
135	paul	1980	antwerp	2006

5 The relational data model

The relational data model is a widely used logical data model. It is used by the major commercial and free DBMS such as Oracle, DB2, SQL Server, Access, MySQL, PostgreSQL.

As in the ER model, two fundamental concepts in the relational model are the database *instance* and the database *schema*. Again, the database schema describes what *type* of information will be stored in the database, whereas the instance is an actual *contents* of the database.

5.1 Relation schemes and instances

In the relational model, all information is stored in the form of tables, called *relations* (not to be confused with the relationships of the ER model!) An example of such a table is shown in Table 1. Note that the columns are named; we call the names of the columns *attributes*. The set of attributes of a relation serves as the *scheme* of that relation.

In database theory, the rows of a table are called *tuples*. But, what exactly is a tuple, formally? For example, consider the first row (tuple) from Table 1:

$$t = (123, \text{john}, 1980, \text{diepenbeek}, 2006)$$

We can view t formally as a *function* from the set of its attributes to the values for the attributes:

$$t = \left\{ \begin{array}{ll} \text{number} & \mapsto 123 \\ \text{name} & \mapsto \text{john} \\ \text{birthdate} & \mapsto 1980 \\ \text{address} & \mapsto \text{diepenbeek} \\ \text{year} & \mapsto 2006 \end{array} \right\}$$

In particular, we have $t(\text{number}) = 123$; $t(\text{name}) = \text{john}$; and so on.

We thus have the following formal definitions:

Definition 3.

- A relation scheme is a finite set R of attributes.
- A tuple of type R is a function $t\colon R \to \mathbb{V}$, where $\mathbb{V}$ is the universe of all possible attribute values.
- A relation instance of R then is a finite set of tuples of type R.

The set $\mathbb{V}$ will typically contain all numbers, all strings, but possibly also other data items, such as dates, images, audio, etc. In principle it is possible to store any data item as the value of an attribute in a tuple in a relation. Note that every tuple $t : R \to \mathbb{V}$ has just the finite set R as its domain. So a tuple is a *finite* function, quite unlike the usual functions from $\mathbb{R}$ to $\mathbb{R}$ that you may be more familiar with.

5.2 Functional dependencies

A relation scheme on itself is almost devoid of any meaning: it is just a set of attributes, and any set of tuples over those attributes is a possible instance. In reality, attributes have an intended meaning, and as a consequence, often not all possible instances will make sense. *Functional dependencies* are a standard formal way of giving some meaning to the attributes in a relation scheme.

Example 15. Consider the following large relation scheme:

$$\{\text{studnr}, \text{sname}, \text{bdate}, \text{bplace}, \text{bcountry}, \text{coursenr}, \text{ctitle}, \text{score}, \text{honor}\}$$

The scheme represents information about students (student number, student name), their birth information (birth date, birth place, birth country), courses they have taken an exam on (course number, course title), the score they received, and the honor deserved by this score. Say that scores are integers between 0 and 20; then a score lower than 10 receives the honor 'insufficient'; between 10 and 13 is 'sufficient'; 14 or 15 is 'distinction'; 16 or 17 is 'great distinction'; and 18, 19 or 20 is 'greatest distinction'.

We now observe that the attributes in this scheme are not all independent. For example, the attribute studnr determines sname, bdate, bplace, and bcountry. In other words, we do not want to allow instances that contain two tuples with the same studnr, but with a different sname, or bdate, or bplace, or bcountry. This integrity constraint is expressed by the following formula, called a functional dependency (FD):

$$\text{studnr} \to \text{sname}, \text{bdate}, \text{bplace}, \text{bcountry}$$

Also other FDs come out naturally in this application:

$$\begin{aligned} &\text{bplace} \to \text{bcountry} \\ &\text{coursenr} \to \text{ctitle} \\ &\text{score} \to \text{honor} \end{aligned}$$

Formally, we define:

Definition 4. Let R be a relation scheme. A functional dependency over R is a formula of the form $X \to Y$, with X and Y subsets of R. A relation instance I of R is said to *satisfy* this FD if I does not contain two tuples that agree on X but differ on Y (i.e., agree on all attributes of X, but differ on at least one attribute of Y).

If Σ is a set of FDs, we say that instance I satisfies Σ if I satisfies every FD in Σ.

Note that an FD of the form $X \to Y$ with $Y \subset X$ is uninteresting because *any* instance satisfies such an FD. (Make sure you see why!) We call such an FD *trivial*.

5.3 The design of relation schemes

When designing an ER schema, we are actually designing various relation schemes. Indeed:

- For each entity type that we design, we specify its set of attributes $\mathcal{A}$, and its key K (so K is a subset of $\mathcal{A}$). Note that $\mathcal{A}$ is nothing but a relation scheme in disguise. Moreover, the key K corresponds to the FD

  $$K \to \mathcal{A}.$$

  But perhaps there are also additional FDs that can be expected to hold in all allowed instances. The designer must carefuly think about these and state them explicitly.

- But also for each relationship type that we design, we implicitly design a relation scheme. Indeed, consider a relationship type R. Let $l_1, \ldots, l_n$ be the list of roles on the edges from R to entity types. Here we are assuming that none of the roles is blank; when a role on an edge is blank, we can always follow the convention of using as a role, the the entity type that is at the head of the edge. Moreover, let $\mathcal{A}$ be the set of attributes of relationship type R. Then the union $\{l_1, \ldots, l_n\} \cup \mathcal{A}$ is a relation scheme. The values for the roles in a tuple would be entity nodes participating in a relationship, and the values for the attributes would be the attribute values for the relationship.

  Then the unique relationship constraint can be formulated as the following FD:

  $$\{l_1, \ldots, l_n\} \to \mathcal{A}.$$

  Moreover, we will see later that functional edges also give rise to FDs!

  But apart from these FDs, the designer should also explicitly state any other FDs over this relation scheme that he expects to be satisfied. This is illustrated in the following example.

Example 16. Reconsider the relationship type *exam* between students and courses, with an attribute score. But suppose (it will later turn out to be a bad idea) we add also the attribute honor, with the same meaning as in Example 15. So, we have roles student and course, and attributes score and honor, so all in all we have the relation scheme

$$\{\text{student}, \text{course}, \text{score}, \text{honor}\}.$$

Due to the unique relationship constraint we state the following FD:

$$\text{student}, \text{course} \to \text{score}, \text{honor}$$

But in addition, we are expected to state that also the following FD will hold (as we already saw in Example 15):

$$\text{score} \to \text{honor}$$

□

Table 2: A relation instance with redundancy and risking update anomalies. ('S' stands for 'sufficient'; 'D' for 'distinction'; and 'GTD' for 'greatest distinction'.)

student	course	score	honor
john	math	12	S
john	french	14	D
mary	math	14	D
mary	french	12	S
paul	latin	19	GTD

Also when we are not using the ER methodology but are designing relation schemes directly from scratch, we should state which FDs we expect to hold in the allowed instances. An example of this was already given in Example 15, although that example is quite extreme; nobody in his right mind would design such a monster of a relation scheme.

5.4 Boyce-Codd normal form (BCNF)

Given a relation scheme with a set of FDs, it will either be a "good" scheme or a "bad" one. Interestingly, it is possible to detect the bad schemes automatically, based on the FDs: a scheme will be "bad" if the set of FDs contains a "bad" FD. Of course we should now define formally what are those "bad" FDs.

Example 17. The relation scheme with the FDs of Example 16 is bad. To see why, consider the instance shown in Table 2. This relation suffers from three classical problems:

Redundancy: That a score of 12 is Sufficient, and that a score of 14 is Distinction, is mentioned twice in the table.

Update anomalies: Because the FD score $\rightarrow$ honor must hold, the system must do a lot of work to check the integrity of updates. For example, if a user tries to update the D in the second tuple to, say, a GD, this would cause a violation of the FD (because the third tuple would then have the same value for score but a different value for honor).

Deletion anomalies: Because there is no student in the instance with a score of 16 or 17, we cannot see from this instance which honor (namely, great distinction) such a score would deserve. Similarly, if there were just one such a student, but the student would be deleted, the honor information would again disappear together with the student.

It is clear that the FD score $\rightarrow$ honor causes all the trouble. The other FD student, course $\rightarrow$ score, honor causes less trouble, because it representes a key: the values for student and course determine all other attributes. Hence, redundancy cannot occur. The system must still check the key when allowing updates, but this is normal because keys must be checked anyway. □

From the above example we conclude that FDs that express keys are OK, but other FDs are bad. This is the essence of Boyce-Codd normal form (BCNF).

But it is not so straightforward to define what it means for an FD to express a key, as the following example shows.

Example 18. If, for the relation scheme $\{\text{student}, \text{course}, \text{score}, \text{honor}\}$ we take the following two FDs:

$$\text{student}, \text{course} \to \text{score}$$
$$\text{score} \to \text{honor}$$

At first sight it seems that the first FD does not express a key, because student and course determine only score but not honor. However, because of the second FD, score determines honor anyway, so we can conclude that $\{\text{student}, \text{course}\}$ is still a key. In other words, *any instance that satisfies the above two FDs, also satisfies the FD*

$$\{\text{student}, \text{course}\} \to \{\text{score}, \text{honor}\}.$$

□

We see from the previous example that we need to be able to determine, from a given set Σ of FDs, which keys, or more generally, which other FDs are *implied* by Σ. This leads to the following formal definition of BCNF:

Definition 5. Let R be a relation scheme and let Σ be a set of FDs over R.

1. Let σ be another FD. We say that Σ *implies* σ if any relation instance that satisfies Σ, also satisfies σ.
2. A set of attributes X is called a *key* with respect to Σ, if the FD $X \to R$ is implied by Σ.
3. We say that the pair (R, Σ) is in Boyce-Codd normal form (BCNF) if the only non-trivial FDs that are implied by Σ have a key as the left-hand side.

Let us see some examples:

Example 19. The scheme and set of FDs from Example 16 is not in BCNF, because the FD score $\to$ honor is clearly implied; as a matter of fact, it is already part of the given set of FDs! And the left-hand side, $\{\text{score}\}$, of this FD is not a key, as we can see, for example, from the instance shown in Table 2.

Example 20. The scheme $\{\text{studnr}, \text{name}, \text{address}, \text{year}\}$ with as only FD:

$$\text{studnr} \to \text{name}, \text{address}, \text{year}$$

is in BCNF because the only non-trivial FDs that are implied by that FD have studnr on the left-hand side, which is clearly a key. Indeed, the FD

$$\text{studnr} \to \text{studnr}, \text{name}, \text{address}, \text{year}$$

is obviously implied by itself (adding attributes from the left-hand side into the right-hand side obviously does not change the meaning of an FD).

Example 21. Consider the scheme $\{A, B, C\}$ with FDs $A \to C$ and $AC \to B$. This is in BCNF because these two FDs together imply $A \to BC$. Hence, A is a key, and certainly AC then is also a key (a superset of a key is always also a key; do you see why?) □

To conclude this section, we stress that it does not make sense to ask whether a relation *instance* is in BCNF; likewise, it is not very interesting to ask whether a relation scheme *without* any FDs is in BCNF. The notion of BCNF is really defined only for pairs (R, Σ) like we have defined it above. (Of course, Σ might be empty, i.e., no FDs are given, and in that case, BCNF trivially holds, but that simply means that we do not know anything about the meaning of the attributes and thus that any possible relation instance is allowed.)

5.5 Implication of FDs

It turns out that we can verify automatically, by an algorithm, whether a given relation scheme with a set of FDs is in BCNF. If we look at Definition 5 of BCNF we see that, in order to check whether (R, Σ) is in BCNF, we need to be able to generate all nontrivial FDs that are implied by Σ. Indeed, once we have done that, we can check that all those FDs have only keys as left-hand sides (and to check wether a left-hand side X is a key, we just check if $X \to R$ belongs to the implied FDs).

Fortunately, there is indeed an algorithm that generates all FDs that are implied by a given set Σ of FDs over R. This is quite cool; it is a form of automated reasoning! The algorithm will actually compute, for each subset X of R, the *closure* of X, denoted by X^+. This closure is the largest set of attributes such that the FD $X \to X^+$ is still implied by Σ. Then, to check whether (R, Σ) is in BCNF, it suffices to compute, for each left-hand side X of some FD from Σ, the closure of X, and check that it is either just X itself or is the whole R. Indeed, if $X^+ = X$, this means that there are no implied FDs with X as left-hand side; if $X^+ = R$, this means that X is a key w.r.t. Σ.

The algorithm for computing the closure of X is as follows:

1. Initialise $X^+ := X$.
2. Repeat until X^+ does not change anymore:
   - For each FD $Y \to Z$ such that Y is a subset of X^+:
     - Add Z to X^+.

One can prove the following theorem:

Theorem 1. *The above algorithm computes X^+ correctly, i.e., X^+ as computed above is indeed the largest set of attributes such that $X \to X^+$ is implied by Σ.*

We will omit the proof.[5]

Example 22. Let us apply the algorithm to the relation scheme and FDs of Example 18. For $X = \{\text{student}, \text{course}\}$, we have:

1. X^+ is initialized as $\{\text{student}, \text{course}\}$.
2. By the FD $\text{student}, \text{course} \to \text{score}$, we get $X^+ = \{\text{student}, \text{course}, \text{score}\}$.
3. By the FD $\text{score} \to \text{honor}$, we get $X^+ = \{\text{student}, \text{course}, \text{score}, \text{honor}\}$; these are already all attributes so the algorithm stops.

[5]The proof can be found in several books, e.g., Ullman and Widom: *A First Course in Database Systems*, or Paredaens et al: *The Structure of the Relational Database Model.*

For $X = \{\text{score}\}$ things are much simpler. By the FD score $\rightarrow$ honor we can add honor to X^+, and then we can add nothing anymore. So $\{\text{score}\}^+ = \{\text{score}, \text{honor}\}$.

Example 23. Let us apply the algorithm to the relation scheme and FDs of Example 21. For $X = \{A\}$, we have:

1. By the FD $A \rightarrow C$, we get $X^+ = \{A, C\}$;
2. Then by the FD $AC \rightarrow B$, we get $X^+ = \{A, B, C\}$ which is everything so the algorithm stops.

5.6 Decomposition of relation schemes

What to do when we find out that the relation scheme we designed is not in BCNF? It means that we made a design mistake, and we must split the relation scheme in two, or perhaps even more, parts.

Example 24. Let us reconsider (Example 16) the relationship type *exam* between courses and students with attributes score and honor. We have seen in the meantime that the resulting relation scheme with FDs is not in BCNF, due to the "bad" FD score $\rightarrow$ honor. This bad FD in fact suggests that we better split the relation scheme in two parts:

- $\{\text{student}, \text{course}, \text{score}\}$ with FD student, course $\rightarrow$ score;
- $\{\text{score}, \text{honor}\}$ with FD score $\rightarrow$ honor.

In our ER design, this would mean that we make a separate entity *score*, with two attributes score and honor, with score being the key attribute. We then transform the relationship *exam* in a ternary relationship between courses, students, and scores. To represent the FD student, course $\rightarrow$ score we make the relationship functional in the role score. □

In general, splitting up a relation scheme with FDs (R, Σ) is called *decomposition* and can be done using the following algorithm:

- If (R, Σ) is in BCNF, the relation scheme is fine and no decomposition is needed.
- Otherwise:
  1. let X be the left-hand side of a bad FD (if there are several bad FD's, just pick one). Then split R in two parts R_1 and R_2, where
     $$R_1 = X^+$$
     $$R_2 = X \cup (R \setminus X^+)$$
     Note that "split" is perhaps a misleading word, because R_1 and R_2 are not disjoint; they have all the attributes from X in common.
  2. We also determine two new sets of FDs: Σ_1 for R_1, and Σ_2 for R_2:
     - Σ_i is the set of all FDs of the form $Y \rightarrow Y^+ \cap R_i$, for $Y \subset R_i$.
  3. Now decompose, if necessary, (R_1, Σ_1) and (R_2, Σ_2) further, using the same algorithm.

Example 25. Let us decompose the relation scheme with FDs from Example 15. So R equals:

$$\{\text{studnr}, \text{sname}, \text{bdate}, \text{bplace}, \text{bcountry}, \text{coursenr}, \text{ctitle}, \text{score}, \text{honor}\}$$

and Σ equals:

$$\begin{aligned}
&\text{studnr} \to \text{sname}, \text{bdate}, \text{bplace}, \text{bcountry}\\
&\text{bplace} \to \text{bcountry}\\
&\text{coursenr} \to \text{ctitle}\\
&\text{score} \to \text{honor}
\end{aligned}$$

1. (R, Σ) is not in BCNF; the first FD is a bad FD, for example (actually, all FDs are bad here). Let us take the first FD to decompose, so $X = \{\text{studnr}\}$. We get

   $$\begin{aligned}
   R_1 &= X^+ = \{\text{studnr}, \text{sname}, \text{bdate}, \text{bplace}, \text{bcountry}\}\\
   R_2 &= X \cup (R \setminus X^+) = \{\text{studnr}, \text{coursenr}, \text{ctitle}, \text{score}, \text{honor}\}
   \end{aligned}$$

   We also determine

   $$\begin{aligned}
   \Sigma_1 = \{&\text{studnr} \to \text{sname}, \text{bdate}, \text{bplace}, \text{bcountry}\\
   &\text{bplace} \to \text{bcountry}\}\\
   \Sigma_2 = \{&\text{coursenr} \to \text{ctitle}\\
   &\text{score} \to \text{honor}\}
   \end{aligned}$$

2. (R_1, Σ_1) is still not in BCNF, with the bad FD bplace $\to$ bcountry. Decomposing on the left-hand side of that FD, so $X = \{\text{bplace}\}$, we get

   $$\begin{aligned}
   R_{11} &= \{\text{bplace}, \text{bcountry}\}\\
   \Sigma_{11} &= \{\text{bplace} \to \text{bcountry}\}\\
   R_{12} &= \{\text{studnr}, \text{sname}, \text{bdate}, \text{bplace}\}\\
   \Sigma_{12} &= \{\text{studnr} \to \text{sname}, \text{bdate}, \text{bplace}\}
   \end{aligned}$$

3. (R_{11}, Σ_{11}) and (R_{12}, Σ_{12}) are now both in BCNF.

4. (R_2, Σ_2) is also not in BCNF, with two bad FDs coursenr $\to$ ctitle and score $\to$ honor. Let us choose to decompose on $X = \{\text{coursenr}\}$; we get

   $$\begin{aligned}
   R_{21} &= \{\text{coursenr}, \text{ctitle}\}\\
   \Sigma_{21} &= \{\text{coursenr} \to \text{ctitle}\}\\
   R_{22} &= \{\text{studnr}, \text{coursenr}, \text{score}, \text{honor}\}\\
   \Sigma_{22} &= \{\text{score} \to \text{honor}\}
   \end{aligned}$$

5. (R_{21}, Σ_{21}) is now in BCNF, but (R_{22}, Σ_{22}) still isn't, with the bad FD score $\to$ honor. Decomposing on $X = \{\text{score}\}$ we get

   $$\begin{aligned}
   R_{221} &= \{\text{score}, \text{honor}\}\\
   \Sigma_{221} &= \{\text{score} \to \text{honor}\}\\
   R_{222} &= \{\text{studnr}, \text{coursenr}, \text{score}\}\\
   \Sigma_{222} &= \varnothing
   \end{aligned}$$

6. (R_{221}, Σ_{221}) and (R_{222}, Σ_{222}) are now in BCNF; note in particular that Σ_{222} is empty, so for R_{222} no FDs are left at all, so it is trivially in BCNF.

To summarize, we have decomposed (R, Σ) in the following parts:

$\{\text{bplace}, \text{bcountry}\}$ with key bplace;
$\{\text{studnr}, \text{sname}, \text{bdate}, \text{bplace}\}$ with key studnr;
$\{\text{coursenr}, \text{ctitle}\}$ with key coursenr;
$\{\text{score}, \text{honor}\}$ with key score; and
$\{\text{studnr}, \text{coursenr}, \text{score}\}$ with no key.

In this example, each BCNF relation scheme ends up with just one key, but there are examples where BCNF relation schemes have several keys.

5.7 Database schemas and instances

So far we have focused on individual relation schemes and instances, but a relational database is a collection of relations. So, in general, a *relational database schema* is a collection of relation schemes, where we also give a name to each relation scheme, for easy reference. Following good practice, each relation scheme should be in BCNF, so there are only keys, no other FDs. The database schema will indicate keys (possibly more than one) for each relation scheme. A relational database *instance* then is a collection of relation instances, one for each relation scheme. Obviously each relation instance must satisfy the key constraints of the relation scheme.

A relation scheme with name N and set of attributes $\{A, \ldots, B\}$ is often denoted as follows:

$N(A, \ldots, B)$

We will use this notation in the sequel.

5.8 Inclusion dependencies

Keys and FDs are integrity constraints that apply to individual relation instances. But there are also an important kind of integrity constraints that apply to two relation instances. These are the inclusion dependencies, and they correspond to the referential integrity constraints we have already seen in the ER model.

Example 26. Consider a database schema with the following relation schemes:

professor(empnr, name, address)
course(coursenr, title)
teaches(c, p)

The attributes 'c' and 'p' in the teaches relation represent, of course, a course number and a professor (more precisely, his empnr). In a datatase instance, we expect that if a course number occurs in the teaches relation, it also occurs in the course relation, and if an empnr occurs in the teaches relation, it also occurs in the professor relation. Such integrity constraints are called *inclusion dependencies*. They are expressed in the following form:

teaches[c] $\subset$ course[coursenr]
teaches[p] $\subset$ professor[empnr]

□

Formally, we define:

Definition 6. An inclusion dependency is a formula of the form

$$N_1[A_1, \ldots, A_k] \subset N_2[B_1, \ldots, B_k]$$

where:

- N_1 and N_2 are the names of two relation schemes in the database schema, with sets of attributes R_1 and R_2 respectively;
- $A_1, \ldots, A_k$ are elements of R_1;
- $B_1, \ldots, B_k$ are elements of R_2.

Note that $A_1, \ldots, A_k$ do not have to be *all* attributes of R_1, just a subset, and likewise for $B_1, \ldots, B_k$ and R_2. Indeed, in many cases, $k = 1$, as in the above Example 26.

Now a database instance I is said to *satisfy* this inclusion dependency if the following holds. Let J_i be the relation instance of N_i, for $i = 1, 2$. Then for every tuple $t_1 \in J_1$, there must be a tuple t_2 in J_2 such that

$$(t_1(A_1), \ldots, t_1(A_k)) = (t_2(B_1), \ldots, t_2(B_k)).$$

When an inclusion dependency as in the above definition is such that $\{B_1, \ldots, B_k\}$ is a key for N_2, the inclusion dependency is also called a *referential integrity constraint*, and $A_1, \ldots, A_k$ is then called a *foreign key* that *references* the key $B_1, \ldots, B_k$.

5.9 Database schemas in SQL

In SQL, the different relation schemes of a database schema are declared using `CREATE TABLE` statements. In that statement one can immediately also sets of attributes to be keys, using the `PRIMARY KEY` or `UNIQUE` clauses. There can be only one primary key, but several additional unique keys, and the meaning is more or less the same; details vary from DBMS to DBMS.

There is also an `ALTER TABLE` statement that allows making changes to the relation scheme.

In SQL it is also possible in the `CREATE TABLE` statement to declare foreign keys, but only if they reference primary keys (not unique keys).

Example 27. Example 26 can be expressed in SQL as shown in Figure 18.

```
create table Professor (
  empnr int not null,
  name varchar(40),
  address varchar(100),
  PRIMARY KEY (empnr)
);
create table Course (
  coursenr not null int,
  title varchar(50),
  PRIMARY KEY (coursenr)
);
create table Teaches (
  c int,
  p int,
  FOREIGN KEY (c) REFERENCES Course(coursenr),
  FOREIGN KEY (p) REFERENCES Professor(empnr)
);
```

Figure 18: Database schema expressed in SQL.

5.10 Column types, large objects, and object-relational

In our theoretical treatment, we have ignored *types* for the attributes (columns) of a relation. As we have already seen in Figure 18, in SQL, one must specify a type for each attribute; then only values of that type can be taken by that attribute. Standard column types of SQL are strings, integers, floating-point numbers, but also dates and times.

It is also possible to store very large objects as attribute values. For example, consider a relation scheme for persons, with attributes name (string), age (integer), and photo (a JPG file). The photo attribute can then be declared as type BLOB (binary large object) and we can store the content of a JPG file as the value of that attribute. Similarly, there is the column type CLOB (character large object) for storing the content of large textfiles. For example, consider a relation scheme for articles with attributes title (string), author (string), and content (CLOB). The content attribute can then contain the entire text of the article.

In *object-relational* databases (already supported by the three big ones Oracle, DB2 and SQL Server) it is even possible for the programmer to define his own column types, much in the same way as he defines Java classes.

6 From ER to Relational

When designing an ER schema, we are designing entity types and relationship types. When doing so, we are effectively designing relation schemes, as we have noted already in Section 5.3: the relation scheme corresponding to an entity type consists of all the attributes of that entity type; the relation scheme corresponding to a relationship type consists of all the roles plus all the attributes of that relationship type. When designing our ER schema, we should make sure to write down, for each obtained relation scheme, the FDs we expect, and in this way make sure that our relation schemes are in BCNF (if necessary, performing decomposition).

After we have done so, we can proceed and create a relational database schema from the ER schema. Moreover, we will formulate key constraints and referential integrity constraints. The purpose of all these constraints is to make sure that only relational database instances are allowed that represent a legal ER instance.

6.1 Mapping entity types

The mapping from ER to relational is straightforward for entity types. For each entity type E, with set of attributes $\mathcal{A}$, and key K, we create a relation scheme with name E and set of attributes $\mathcal{A}$. We designate K to form the primary key, but if from the BCNF design also other keys result, we declare these additionally as unique key constraints.

6.2 Mapping relationship types

The mapping of relationship types is more complicated. We first see two examples.

Example 28. Consider entity types Student(<u>name</u>, <u>bdate</u>, <u>bplace</u>, address, year) and Course(<u>coursenr</u>, ctitle). Consider the relationship type *exam* between students and courses with attribute score. We map this relationship type to a relation scheme that contains the key attributes for the participating entity types, plus the attribute score:

$$\text{Exam}(\text{name}, \text{bdate}, \text{bplace}, \text{coursenr}, \text{score})$$

We call the attributes name, bdate and bplace the *role attributes* for role Student, and we call the attribute coursenr the role attribute for role Course.

To enforce the unique relationship constraint, we declare the set of all role attributes together to be a unique key:

```
UNIQUE (name, bdate, bplace, coursenr)
```

Furthermore, the role attributes for each role separately are clearly a foreign key to the relation corresponding to that role:

```
FOREIGN KEY (name,bdate,bplace)
  REFERENCES Student(name,bdate,bplace),
FOREIGN KEY (coursenr) REFERENCES Course(coursenr)
```

Example 29. Consider an entity type

$$\text{Person}(\underline{\text{name}}, \underline{\text{bdate}}, \underline{\text{bplace}}, \text{address}, \text{weight})$$

and a relationship type *father* between two persons (once in role child, once in role father). The role attributes for role child are name, bdate and bplace, but the role attributes for role father are the same! This is a problem as we cannot have the same attributes twice in a relation scheme. We can solve this problem, however, by appending the name of the role to the attribute names:

$$\text{Father}(\text{name\_child}, \text{bdate\_child}, \text{bplace\_child}, \text{name\_father}, \text{bdate\_father}, \text{bplace\_father})$$

We have again two referential integrity constraints:

```
FOREIGN KEY (name_child,bdate_child,bplace_child)
  REFERENCES Person(name,bdate,bplace),
FOREIGN KEY (name_father,bdate_father,bplace_father)
  REFERENCES Person(name,bdate,bplace)
```

Now note that the father role naturally is functional in this relationship. We can enforce this cardinality constraint by declaring the following unique key:

```
UNIQUE (name_child,bdate_child,bplace_child)
```

Example 30. Consider entity types

$$\text{Professor}(\underline{\text{empnr}}, \text{salary}) \text{ and } \text{Course}(\underline{\text{coursenr}}, \text{ctitle})$$

and a relationship type *teaches* between professors and courses. We map this to the relation scheme Teaches(empnr, coursenr). If we declare the Professor role to be functional (requiring that every course is taught by at most one professor) then we can, as in the previous example, express this as a key constraint:

```
UNIQUE (coursenr)
```

Additionally, however, we also would like that every course is taught by someone. Such a requirement cannot be expressed in the form of a key. It can be expressed as an inclusion dependency:

$$\text{Course}[\text{coursenr}] \subset \text{Teaches}[\text{coursenr}]$$

Since coursenr is not a key for the Teaches relation, however, this inclusion dependency cannot be directly expressed in SQL as a foreign key constraint. More powerful ways exist in SQL to assert general inclusion dependencies and many other kinds of integrity constraints, through the ASSERT statement; we refer to an SQL handbook for more information. □

We can now describe the general mapping of relationship types to relation schemes as follows. Let R be a relationship. Let $e_1, \ldots, e_n$ be the edges from R to entity types $E_1, \ldots, E_n$, respectively, and let $l_1, \ldots, l_n$ be the roles on these edges; as before we can safely assume that no role is blank. Let $\mathcal{A}$ be the set of attributes of R. For each i, let K_i be the key of E_i.

The relation scheme for R: For each edge e_i we construct the following set of attributes:

$$\mathcal{A}_i = \{l_i\_A \mid A \in K_i\}$$

That is, we prepend the role to each key attribute of entity E_i. This was already illustrated in Example 29.

We now create a relation scheme with name R and the following set of attributes:

$$\mathcal{A} \cup \mathcal{A}_1 \cup \cdots \cup A_n$$

The integrity constraints for R:

1. Unless $\mathcal{A}$ is empty, we must enforce the unique relationship constraint by declaring $\mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n$ to be a key.

2. Furthermore, for each edge e_i that is functional, we declare the set

   $$\mathcal{A}_1 \cup \cdots \cup \widehat{\mathcal{A}_i} \cup \cdots \cup \mathcal{A}_n$$

   to be a key also. Here, $\widehat{\mathcal{A}_i}$ signifies that we *omit* the attributes from $\mathcal{A}_i$.

3. Moreover, for each edge e_i, we declare the following referential integrity constraint. Let $K_i = \{A, \ldots, B\}$. Then we declare the inclusion dependency:

   $$R[l_i\_A, \ldots, l_i\_B] \subset E_i[A, \ldots, B]$$

7 The relational algebra

When we want to query a relational database, we formulate our query in a query language. *In relational databases, the answer to a query is always again a relation.* We see why this is natural by consider some example queries:

1. List all students that are enrolled for course C1234; for each such student, list only the student number and the address. The answer to this query is clearly again a relation over the attributes studnr and address.

2. List all professors that do not teach any course; for each such professor, list their employee number and their salary. Again the answer is a relation over the attributes empnr and salary.

3. Give the year of study of the student with number 08123456. Although the answer to this query is just a single number (a year), we can still view it as a relation over the attribute year, a relation that contains just a single tuple. Actually, it could even be that there is no student with number 08123456; in that case, the answer would be empty, which is still a relation (namely, the empty set of tuples).

The most popular query language for relational databases is SQL. We will learn about SQL later, but it is also important to have an idea of what the DBMS must actually do to answer an SQL query. Roughly, the DBMS will have to perform the following tasks:

1. Receive the query through some connection between the user (application program) and the DBMS.

2. Parse the SQL expression of the query.

3. Translate the SQL expression in a query plan: this is a concrete strategy to compute the answer to the query. This strategy is typically composed of many operations that are applied to relations. Each operation results in a new relation, to which additional operations are applied, and so on, until a final relation is obtained that is the answer to the query.

4. Return the tuples of the answer relation to the user.

In this chapter we will have a look at the operations on relations that are applied in query plans (step 3 above). Surprisingly, it turns out that the answer to any SQL query can always be computed by a composition of just seven basic operations:

1. Selection

2. Projection

3. Join

4. Renaming

5. Union

6. Difference

7. Aggregation

These seven operators form what is known as the *extended relational algebra*. The 'extended' refers to aggregation: the first six operators form the plain relational algebra.

Before we look more closely to the relational algebra, we give two remarks.

1. There is a similarity here with arithmetic. We know that any elementary calculation on numbers can be computed by composing the basic arithmetic operators of addition, subtraction, multiplication, and division. A DBMS does the same but with entire relations: any SQL query can be computed by composing the basic operators on relations of the relational algebra.

2. When writing an SQL query, we do not have to worry about how the answer can be computed in the relational algebra. This is the (large) task of the DBMS; the DBMS will find an appropriate relational algebra calculation, and it will even choose among different possible calculations the one that can be most efficiently computed. Therefore we say that SQL is a "declarative" query language: the user only has to specify what he wants, but he does not have to specify how the answer should be computed.

   This issue will, of course, become more clear when we look in more detail to SQL.

7.1 Selection

The selection operator, denoted by the symbol σ, takes a condition θ as parameter. The condition θ should be a condition about tuples: some tuples satisfy θ, others don't. The selection then selects those tuples from its input relation that satisfy θ. Formally:

$$\sigma_\theta(r) = \{t \in r \mid t \text{ satisfies } \theta\}$$

Example 31. Consider the query "give me all students that started their studies in the year 2000 or later". We can express this query as a selection $\sigma_{\text{year}\geq 2000}(\text{Student})$, where we use the name Student to refer to the current instance of the student relation. The result will be the set of all tuples t in the student relation instance for which $t(\text{year}) \geq 2000$. If there is no such student in our database (i.e., no such tuple in our relation instance), the result will be the empty set.

Example 32. Selection conditions can also involve several attributes. Suppose we want all students that were younger than 20 when starting their studies. Assuming attributes year (first year of study) and byear (birth year), we can express this query by the selection $\sigma_{\text{year}<\text{byear}+20}$.

7.2 Projection

Often we do not want all attributes in the answer, but only the attributes we are interested in. The projection operator keeps only some of the attributes from its input relation; we say that we *project* on those attributes. The operator is

denoted by π and takes as parameter a set Z of attributes. Applied to any relation instance r whose scheme includes Z, we have:

$$\pi_Z(r) = \{t|_Z \mid t \in r\}$$

Here, $t|_Z$ denotes the restriction of t to Z (recall that formally, tuples are functions on a finite domain of attributes, so we can restrict them to a subset of the attributes).

While the selection always returns a relation instance of the same scheme as the input relation, the projection π_Z always returns an instance of scheme Z.

Example 33. Suppose we want to know the name and the address of the student with number 08123456. If we just select $\sigma_{\text{number}=08123456}(\text{Student})$, we get all attributes of that student. To restrict to just name and address, we can perform a projection after the selection:

$$\pi_{\text{name,address}}(\sigma_{\text{number}=08123456}(\text{Student}))$$

7.3 Natural Join

Selection and projection are applied to a single relation instance. We also say that selection and project are *unary* operators. The natural join operator, in contrast, is a *binary* operator: it is applied to two relations and is used to compose two relations. The operator is denoted by the symbol $\bowtie$. The proper name of $\bowtie$ is "natural join" but we will often just say "join".

Example 34. If we want the course numbers of the courses taught by the professor with employee number 8226, we can write

$$\pi_{\text{coursenr}}(\sigma_{\text{empnr}=8226}(\text{Teaches})).$$

But what if we want the titles of those courses? The Teaches relation only gives us the course numbers; the course titles can be found in the Course relation. To connect course numbers in the Teaches relation to course numbers (and titles) in the Course relation, we use the join operation. It is illustrated in Figure 19. Our query then becomes

$$\pi_{\text{ctitle}}(\sigma_{\text{empnr}=8226}(\text{Teaches} \bowtie \text{Course})).$$

But note that we can write the same query also as follows:

$$\pi_{\text{ctitle}}(\sigma_{\text{empnr}=8226}(\text{Teaches}) \bowtie \text{Course}).$$

The difference is that the selection is done before the join, instead of afterwards. This shows that, for a given query, there are many different expressions in the relational algebra for that query. A user does not have to worry about this, as the DBMS will choose the most efficient one (in the example here, doing the selection before the join would be more efficient).

To continue, suppose we now want the titles of all courses taught by the professor named Jones. The Teaches relation only contains employee numbers; the names of professors are in the Professor relation. So now we have to join three relations; we also have to connect the employee numbers in the Teaches relation with the employee numbers (and professor names) in the Professor relation. Our query then becomes:

$$\pi_{\text{ctitle}}(\sigma_{\text{name}=\text{`Jones'}}(\text{Professor}) \bowtie \text{Teaches} \bowtie \text{Course})$$

Teaches

empnr	coursenr
8226	123
8226	114
7654	123
7654	135
9876	126

Course

ctitle	coursenr
math	123
french	114
databases	135
statistics	126

Teaches ⋈ Course

empnr	coursenr	ctitle
8226	123	math
8226	114	french
7654	123	math
7654	135	databases
9876	126	statistics

Figure 19: Illustration of the join operation. For the sake of illustration, we allow here two professors teaching the same course (math 123).

Example 35. For another example, consider the relations Person(id, name, address) and Student(id, year), where Student **isa** Person, so we have the inclusion dependency Student[id] $\subset$ Person[id]. If we want the list of all names of students, this can be computed as

$$\pi_{\text{name}}(\text{Student} \bowtie \text{Person}).$$

□

Formally, consider two relation schemes R_1 and R_2, and let r_1 and r_2 be instances of R_1 and R_2 respectively. Then we define:

$$r_1 \bowtie r_2 = \{t_1 \cup t_2 \mid t_1 \in r_1,\ t_2 \in r_2,\ \text{and } t_1|_{R_1 \cap R_2} = t_2|_{R_1 \cap R_2}\}.$$

So, we take all pairs of tuples that agree on the common attributes of R_1 and R_2, and we join all these pairs of tuples. The result is a relation instance of the scheme $R_1 \cup R_2$.

The result of a join can be much larger than the relations that are being joined. We did not see that effect in Figure 19, because there the common attribute, coursenr, was a key in the Course relation. But in Figure 20 we see that when r_1 has n tuples and r_2 has m tuples, in the worst case the size of $r_1 \bowtie r_2$ can be as large as $n \times m$.

7.4 Cartesian product

If we join two relations that do not have any common attributes, we join *all* possible pairs of tuples from both relations: indeed, the condition that they should agree on the common attributes now becomes void. A simple illustration is given in Figure 21. Note that when we join two relations without common attributes, one relation with n tuples and the other with m tuples, the join will always result in $n \times m$ tuples.

r_1

A	B
a_1	b
a_2	b
$\vdots$	
a_n	b

r_2

B	C
b	c_1
b	c_2
$\vdots$	
b	c_m

$r_1 \bowtie r_2$

A	B	C
a_1	b	c_1
	$\vdots$	
a_1	b	c_m
	$\vdots$	
a_n	b	c_1
	$\vdots$	
a_n	b	c_m

Figure 20: Large join result.

r_1

A	B
a	b
c	a

r_2

C
d
e

$r_1 \bowtie r_2$

A	B	C
a	b	d
a	b	e
c	a	d
c	a	e

Figure 21: Join of two relations without common attributes: all possible pairs of tuples are joined.

The *Cartesian product* is a special kind of join where we explicitly force that there are no common attributes. This is done by renaming each attribute A of a relation named R to $R.A$. The Cartesian product is denoted by the symbol $\times$. Figure 22 shows an illustration.

Why would we ever want to do $\times$ instead of $\bowtie$? One typical example is when we do not want to equate common attributes, but do some other kind of comparison, as shown in the following example.

Example 36. Consider two relations A(item, weight) and B(item, weight). Both relations contain information about items and their weights. (The two relations could come from two different sites.) We now ask for all pairs of items from A and B that weigh together more than 1000:

$$\pi_{\text{A.item,B.item}}\, \sigma_{\text{A.weight+B.weight}>1000}(\text{A} \times \text{B})$$

Queries of this kind are known as "theta-joins", because they still do some kind of join, but with a join condition (referred to as theta) that is different from just agreeing on common attributes. For example, here, theta is the condition A.weight + B.weight $>$ 1000.

7.5 Renaming

Sometimes we need to give a relation a new name. This can be done with the renaming operator, which is denoted by ρ. The operator takes as parameter a new relation name N; when applied to a relation instance r, the result of $\rho_N(r)$ is the same instance, but it now has name N.

Renaming is mostly useful when we want to join a relation with itself, as shown in the following example.

Example 37. Consider the relation Father(child, father). Suppose we want to know, for each child, his grandfather. This query can be expressed using renaming as follows:

$$\pi_{\text{F.child,G.father}}\, \sigma_{\text{F.father=G.child}}(\rho_{\text{F}}(\text{Father}) \times \rho_{\text{G}}(\text{Father}))$$

Note the use of renaming so that the attributes of the two different copies of the Father relation are not mixed up. □

Note that this example query is again a kind of theta-join, but the join condition is an equality. Such theta-joins are called "equi-joins". A natural join is actually a special kind of equi-join, where the equality must hold for the common attributes.

Renaming of individual attributes Instead of given a relation a new name, we can also rename individual attributes. To denote the renaming of attribute A to B (of course B should be a new attribute, not yet in the relation scheme), we write $\rho_{A/B}$. For example, we could have written the grandfather query also as follows:

$$\pi_{\text{child,grandfather}}(\text{Father} \bowtie \rho_{\text{child/father}}\, \rho_{\text{father/grandfather}}(\text{Father}))$$

Teaches

empnr	coursenr
8226	123
8226	114
7654	123
7654	135
9876	126

Course

ctitle	coursenr
math	123
french	114
databases	135
statistics	126

Teaches × Course

Teaches.empnr	Teaches.coursenr	Course.ctitle	Course.coursenr
8226	123	math	123
8226	123	french	114
8226	123	databases	135
8226	123	statistics	126
8226	114	math	123
8226	114	french	114
8226	114	databases	135
8226	114	statistics	126
7654	123	math	123
7654	123	french	114
7654	123	databases	135
7654	123	statistics	126
7654	135	math	123
7654	135	french	114
7654	135	databases	135
7654	135	statistics	126
9876	126	math	123
9876	126	french	114
9876	126	databases	135
9876	126	statistics	126

Figure 22: Cartesian product. Note that coursenr is no longer a common attribute because it is renamed to Teaches.coursenr in the Teaches relation, and to Course.coursenr in the Course relation. So, there are no common attributes and all possible pairs of tuples are joined.

7.6 Union and difference

The union and difference operators: $\cup$ and $\setminus$, are just the well known set-theoretic operators.

Example 38. Recalling the relations A(id, weight) and B(id, weight) from Example 36, suppose we ask for all items in A that weigh less than 1000, plus all items in B that weigh more than 2000. We can express this query using union:

$$\sigma_{\text{weight}<1000}(\text{A}) \cup \sigma_{\text{weight}>2000}(\text{B})$$

Example 39. Given relations

$$\text{Professor}(\text{empnr}, \text{name}) \quad \text{and} \quad \text{Teaches}(\text{empnr}, \text{coursenr}),$$

suppose we ask for all professors that do not teach any course. We can express this query using difference:

$$\pi_{\text{empnr}}(\text{Professor}) \setminus \pi_{\text{empnr}}(\text{Teaches})$$

This query returns the employee numbers of those professors; if we want their name, we have to join again with Professor:

$$\pi_{\text{name}}(\text{Professor} \bowtie (\pi_{\text{empnr}}(\text{Professor}) \setminus \pi_{\text{empnr}}(\text{Teaches}))$$

7.7 Aggregation

An *aggregate function* is a function that works on a bag of values, and returns a single value. Typical examples are:

- The sum (SUM), or average (AVG), of a bag of numbers;
- The minimum (MIN), or maximum (MAX), of a bag of ordered values (such as numbers, but also dates for example);
- The count (COUNT) of a bag (i.e., the number of elements).

Here, by a "bag" we mean a set with duplicates allowed. For example, the average of the bag $\{1, 1, 2, 2, 2, 4\}$ equals 2.

Using the aggregation operation, denoted by γ, we can apply aggregate functions to groups of tuples in a relation. We first see an example.

Example 40. Given a relation Exam(student, course, score), we want to know, for each student, the average score. This can be expressed as an aggregation

$$\gamma_{\text{student;AVG(score)/average}}(\text{Exam}).$$

The idea is that all tuples with the same student value are grouped together; then for each group, the average of the bag of scores is computed; that average becomes the value of a new attribute 'average'. So the resulting relation has scheme {student, average}.

Sometimes we do not want to group but simply want to compute an aggregate function over the entire relation. Suppose we want the maximum score that any student achieved on any exam. This can be expressed as

$$\gamma_{\text{MAX(score)/themax}}(\text{Exam}).$$

Because there is not grouping, the result of this aggregation is a single value, or more precisely, a relation with a single column 'themax' and a single tuple. □

In general, the aggregation operation, denoted by γ, takes four parameters:

1. a set G of attributes, called the grouping attributes;
2. the name of an aggregate function f;
3. an attribute A not in G, called the aggregation attribute; and
4. a new attribute B, called the output attribute.

We denote these parameters by $\gamma_{G;f(A)/B}$. Aggregation is a unary operator: it applies to a single relation instance r. The effect of $\gamma_{G;f(A)}(r)$ is the following:

1. First, the tuples in r are grouped according to G. All tuples of r that agree on all grouping attributes belong to the same group.
2. Then, for each group, we consider the bag of A-values of all tuples in the group. We apply f to this bag.
3. For each group, we output a tuple consisting of the values of the grouping attributes, plus the result of $f(A)$ from the previous step as the value of the output attribute B.

So, the result relation scheme is $G \cup \{B\}$. Note that if G is empty, there is effectively just one large group, namely, the whole relation.

One can also consider aggregation operations that apply multiple aggregate functions. We will just give an example:

Example 41. For relations Student(name, age) and Exam(name, course, score), we want, for each course, the average score, and also the youngest age of the students who took that course. We can express this query as follows:

$$\gamma_{\text{course;AVG(score)/average;MIN(age)/minage}}(\text{Student} \bowtie \text{Exam})$$

7.8 Generalized projection

Now that we have seen that we can perform calculations over bags of attribute values, it seem natural to also want to be able to perform more simple calculations on individual attribute values. This can be done by generalizing the projection operation, to allow not only projection on certain attributes, but also the creation of new attributes based on a calculation performed on existing attributes. We will just give an example.

Example 42. Given a relation Student(name, byear, year) with attributes name, birth year and start year of study, we want to calculate for each student the age at which he started to study. We can express this query as a generalized projection:

$$\pi_{\text{name,age=year}-\text{byear}}(\text{Student})$$

The result is a relation with attributes name and age.