The Relational Model

1. Basic concepts

Databases are born to store huge quantity of data and to fulfil precise requirements of information management, which are:

- *Logical organization of stored data;*
- *Minimal (if not null) amount of data duplication;*
- *Data security and persistency;*
- *Speed of data retrieval;*
- *Synchronous access to the data;*
- *Access tracking.*

In order to assure most of this objective, *IBM developed the Relational model for Data Base (RDB)* management in the early 70s. *This model organizes data taking advantages of concepts of relational algebra* and, by doing so data are organized with a *high level of abstraction* characterized by: *(i) mathematical rigor, (ii) easiness of use and (iii) compliance to the mathematics of sets*.

Although this may seem very complicated, the success of the model lies exactly in its **strong logical abstraction** that makes it possible to use only the concepts of **entities** and **relationships** to build all the entities and the operations needed to physically implement a database. Very briefly, the model allows one to represent data and the relationships among them and, by doing so, it simplifies all the logical operations (i.e., data storing, data retrieval, data aggregation, etc.) needed to convert data into usable information.

Before entering into further details, we note that any RDB must include *three basic parts*:

1. ***Data Definition Language (DDL)*** - *The structural part of the database*, which is a collection of objects and of types needed to build the database. We could say that the DDL defines the pillars and the bricks needed to build the RDB.
2. ***Data Manipulation Language (DML)*** - *The manipulative part of the RDB*, needed to perform all the actions needed to store, search, update and delete the data stored in the physical structure of the RDB.
3. ***Referential Integrity Constraints (RIC)*** - *The part delegated to perform logical checks* concerning the integrity and the correctness of data stored in the RDB. Specifically, the RIC is a collection of rules that ensure data validity and consistency.

In order to thoroughly explain this model, we will take as a reference a simple RDB, designed for the management of a plant shop. The Entity-Relation scheme of the DB (created in Microsoft Access 2010) is shown in Figure 1.1.

Without going into excessive details - everything will become clear later on - the RDB is composed of seven entities (better denoted as Meta-Data or Tables), that are:

- CUSTOMERS;
- EMPLOYEES;
- SHIPPERS;
- PRODUCTS;
- CATEGORIES.
- ORDERS;
- ORDERS DETAILS.

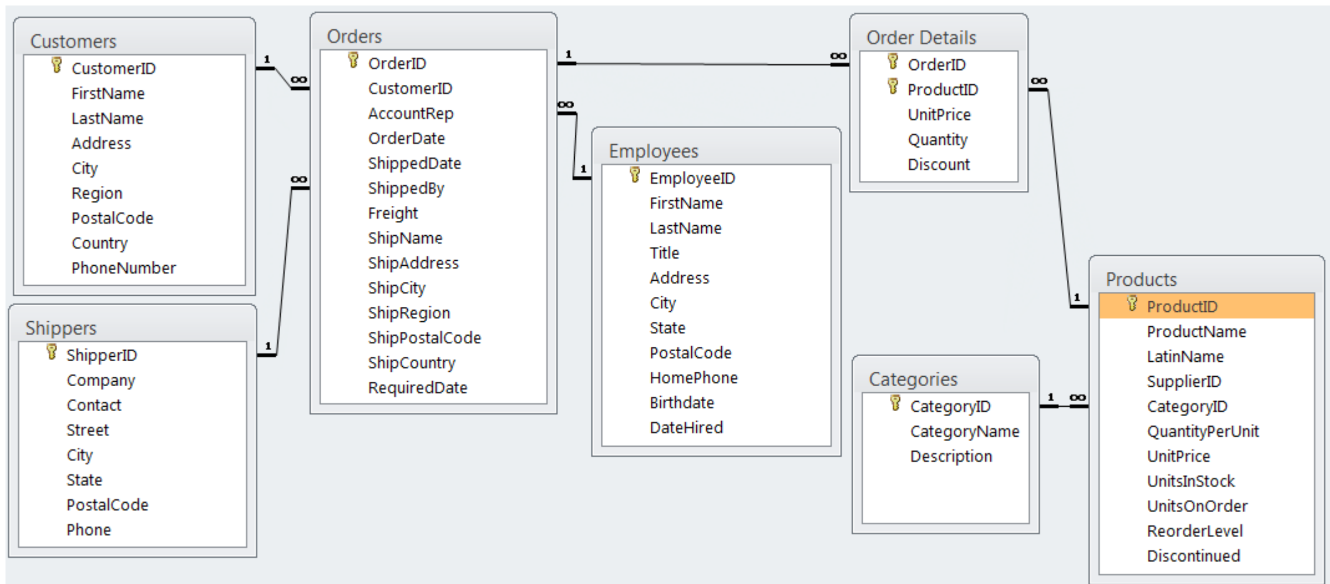

Fig. 1.1 *A basic example of RDB*

The first four Tables are self-explaining as they correspond to physical entities; conversely, the last two deserve some further comments.

The CATEGORIES table allows one clustering products in groups with similar characteristics. For instance, products as 'roses', 'violets', 'cyclamens', 'orchids', 'lilies', 'tulips', etc., belong to the products' category 'Flowers'. Similarly, 'hoes', 'fertilizers', 'watering cans', 'rakes', etc. belong to the products' category 'Gardening Tools'.

Concerning the table ORDER DETAILS, let us start by noting that, as it is obvious, an order is made by a customer, processed by an employee and delivered by a shipper. This explains the three relationships (graphically shown by means of a black connection) between CUSTOMERS, SHIPPERS and EMPLOYEES. Also, and perhaps more important, an order generally contains more than a single product (i.e., a customer could have ordered, at the same time, a bunch of roses, a bunch of orchids and a bucket). Furthermore, the same product type could have

been ordered more times (i.e., many customers could have bought a bunch of roses). Thus, to properly detailing these relationships among the ORDERS and PRODUCTS one has to introduce and additional and virtual entity called ORDER DETAILS. This table contains additional data needed to specify the composition of an order (i.e., products and quantities) and some additional data that directly depend on the order's configuration, such as unit price and discount rate. Actually, the unit price may depend on the order configuration (a bunch of roses alone may cost more than a bunch of roses bought together with a bunch of orchids) and the discount rate may depend both on the number of ordered items and on the customer itself (i.e., recurrent customers may benefit of special offers and extra discounts).

We also note that, in this simplified RDB, there is not a table called STOCK, yet knowing what is available in stock is essential to manage the plant shop. This is because the hypothesis is made that all products are stocked in a single warehouse. For this reason, all data concerning the stock's inventory (such as the Inventory Position, the On Hand the On Order, the Replenishment Lead Time and the Reorder Level) may be included in the PRODUCT table.

2. Relationships among data

In a RDB, data representation is based on the use of **<u>relationships among data and among metadata</u>**.

Specifically, the concept of *<u>algebraic relationship</u>* (among data) can be defined as follows:

«Given a number of 'n' set D_1, D_2, ..., D_n, each one containing a finite number of elements, a relation R among these sets is a set of n-tuples $< d_1, d_2, ..., d_n >$, where d_i is an element belonging to D_i»

<u>The 'n' sets D_1, D_2, ..., D_n are called domains, their names are called attributes and the value 'n' is called the degree of the relation R.</u>

To clarify this concept let us consider four domains:

- NAMES {Bill, Paul, George},
- SURNAMES {Peck, Spark, King},
- AGES {1, 2, 3, …, 100},
- CITIES {London, Cardiff, Liverpool, Bristol}.

Since we have four domains, a relation among these domains will be made of *four-dimensional tuples*, such as:

$$< \text{Bill, Peck, 24, Cardiff} >$$

By aggregating these four data using this relation we obtain the description of a physical entity, i.e., Mr. Bill Peck, of age 24 living in Cardiff. In other words, *<u>a relation among data contextualizes atomic data</u>*, turning them into *<u>usable information</u>*.

More specifically, this relation generates a Metadata or a table, which describes people that could be the employees or the customers of the shop. It is obvious that, at most, the above-mentioned relation can contain

360 (i.e., 3×3×10×4) 4-tuples of the kind < Name, Surname, 24, Cardiff >. However, there is not the necessity to generate them all. For instance, if there is not a Mr. Bill Peck of age 24 living in London who is a customer of the shop, then the 4-tuple <Bill, Peck, 24, London> will not be included in the relation.

Let us make a second example related to the conceptual diagram of Fig. 1.1; to this aim let us consider two domains: Category_ID = {1, 2, 3, 4, ..., 22} and Category_Name = {Bulb, Cactus, Climbers, Flowers, ..., Shrubs}. The ID is a univocal identifier and so, in this case, since each code corresponds to a single name, the only relation R which can be defined between the two domains is the following one:

< (1, bulb), (2, Cactus), (3 Climbers), (4 Flowers), ..., (22, Shrubs) >

Thus n = 2 and the relation corresponds to a set of twenty-two 2-tuples.

As already mentioned, it is *possible and very convenient to represent a relation as a table*, where each row is divided into fields or attributes, each one containing a specific data selected within a specific domain of definition (integer, text, image, date, time, list of objects, etc.).

In summary we can say that **Metadata are the result of algebraic relation among data** and that, at least visually, **Metadata corresponds to Tables**.

As a results concepts of relational algebra may be substituted by the following more familiar terms:

- *Algebraic Relation = Table = Meta Data*
- *n-tuples = Lines = Record*
- Attributes = Columns

Table 1.1 shows an example of what just said.

Tab. 1.1

An algebraic relation may be seen as a Table

Category ID	Category Name	Description
2	Cacti	Indoor cactus plants
3	Ground covers	Herbaceous perennials, evergreen and deciduous shrubs, ivy, vines, mosses
4	Grasses	Lawn grasses for cool climates
5	Flowers	A wide variety of flowers
6	Wetland plants	Plants suitable for water gardens and bogs
7	Soils/sand	Potting soils, peat moss, mulch, bark

It is obvious that the number of columns of a Table corresponds to the degree of the corresponding relationship, which is equal to the number of attributes considered. Conversely, the number of n-tuples (or the number of rows) depends on the number of combinations that are possible between all the elements of the various domains. Let us consider Table 1.1 as an example. In this case, since the relationship among ID, Category_Name and Description is unique, the number of columns equals 22 that is, exactly, the cardinality (i.e., number of elements) of each domain. Obviously, this is the smallest possible value; at the opposite extreme there is the

case where each element of each domain can be associated with all the other ones. If so, if all the *'n'* domains have the same number of elements 'm' then the relation would generate a table with (m^n) columns.

Apart from these unimportant technical aspects, it is extremely important to highlight the fact that, *two fundamental consequence* derive from the above-mentioned definition:

1. *The order of the rows in a table is not significant;*
2. *Each row of a table must be unique* (i.e., at least the value of a field must differ from all the other rows).

The second property is very important as it defines *a sort of hierarchy among the fields* contained in a table. Indeed, since two equal rows cannot be present in the same Table (otherwise the information would be redundant and duplicate), some fields must be unique and, in this sense, they can be considered as more important than the other ones. Thus, it **must be possible (and, indeed, it is in practice) to identify, in each table, a set of fields or attributes (in some circumstances just one) sufficient to uniquely identify each single row**. We will refer to this set of columns (that is defined during the creation of the logical schema) as the **Primary Key (PK)** of the table. As we will see, the PK is essential to avoid data redundancy, to ensure data integrity and, most of all, to establish logical relationships among tables.

In practice, the **PK is the element that ensures the "*traceability*"** of each single line. Let us consider the CATEGORIES_ID table once again. In this case, each one of the first two columns may have been designated as PK, since none of them has duplicate values. In this peculiar case the choice is arbitrary and, of course, due to convenience, we will use the identifier ID (that is a progressive numeric field) as PK.

However, when dealing with more complex tables (obtained from non-unique relations performed on many of the initial data), *selecting a field as PK may not be an arbitrary choice and so the use of an additional ID filed may be strictly necessary*. Let us consider, for instance, the CUSTOMER table. In this case neither the city nor the name of an employee is enough to identify a specific customer (i.e., a row). These data are, in fact, repeated several times within the table: there may be more customers with the same name and, similarly, many customers can live in the same city. Please note that this does not imply a redundancy of information: **some fields may be repeated, but what cannot be repeated is an entire row**. This is the essential point.

Returning to the definition of a PK for the CUSTOMER table, an option could be the use of the surname or, even better, the couple of columns Name and Surname. However, even in this case it is not possible to exclude possible homonyms and, also, the use of two separate columns could create referential integrity problems if the Data Base and the structure of the tables should be changed. It is then a good idea to introduce an ID field containing a different value for each record in the table. Most of the times, this is the simplest, yet most powerful solution. The ID field can contain any type of data, provided that it is unique. In this regard, we can say that commonly the ID field contains a progressive number; sometimes it contains a more comprehensible (readable) alfa-numerical code, as the one shown in Table 1.2.

Tab. 1.2
CUSTOMERS Table

CustomerID	FirstName	LastName	Address	City	Region	PostalCode	Country
ACKPI	Pilar	Ackerman	8808 Backbay S	Bellevue	WA	88004	USA
ADATE	Terry	Adams	1932 52nd Ave.	Vancouver	BC	V4T 1Y9	Canada
ALLMI	Michael	Allen	130 17th St.	Vancouver	BC	V4T 1Y9	Canada
ASHCH	Chris	Ashton	89 Cedar Way	Redmond	WA	88052	USA
BANMA	Martin	Bankov	78 Riverside Dr.	Woodinville	WA	88072	USA
BENPA	Paula	Bento	6778 Cypress Pl	Oak Harbor	WA	88277	USA
BERJO	Jo	Berry	407 Sunny Way	Kirkland	WA	88053	USA
BERKA	Karen	Berg	PO Box 69	Yakima	WA	88902	USA

Later on - talking about relations among tables - we will see that, in some cases, a single field may not be enough to define a PK and that it could be necessary to combine two or more fields to generate a unique identifier. To avoid potential problems of referential integrity, whenever possible, the use of a single field as PK is highly recommended; yet sometimes there are not alternatives and two or more fields must be used.

3. Relationships among tables

Until now we have only considered relations among data, a concept that has brought us to the definition of metadata or tables. This is only the first step in the definition of a RDB; indeed, the **power of such system relays in the possibility to create logical connection among two or more tables**. This requires the definition of relationships between couple of tables; relations that are obtained by connecting together their PKs and by appropriately configuring the referential integrity rules.

Essentially relationships can be of two main types:

- ***One To Many relationship (OTM);***
- ***Many To Many relationship (MTM).***

Let us consider table A and table B. If there is a OTM relation between A and B, then (almost) each record of A has a direct link with a variable number of records of B, whereas, (almost) each record of B has a direct link with a single record of A. In other words, A descend from B and so, B could be named as the *"Father Table"* and A could be named as the *"Son"* one.

Conversely, in case of a MTM relation between A and B, there will be multiple links (among records of A and B) in both directions; for this reason, a MTM relation can be thought as the combination of two distinct OTM relations.

Before proceeding further on, we finally observe that two additional relations belong to the class of the OTM relations. These are:

- *One To One relation (OTO);*
- *Self-Referencing Relation (SRR).*

OTM Relations

An OTM relation takes place every time that a record (or better a metadata) has a link with a variable number of records of another table.

Let us consider, for instance, the PRODUCTS and the CATEGORIES tables. As discussed in Section 1, many products may belong to the same category: 'agave', 'aloe' and 'cactus mouse tail' are all plants belonging to the 'succulent plants' category. Thus, it would be a nonsense sense trying to combine in the same table all the information concerning a plant and the category to which is belongs to. This would inevitably generate data redundancy as clearly shown in the (wrong) example of Table 1.3.

Tab. 1.3
An example of a wrong aggregation of data in a single table

ID	Name	...	Category	Description
1	Aloe	...	Succulent	Plants leaving in dry, hot and sunny places. They need little water.
2	Mouse Tail	...	Succulent	Plants leaving in dry, hot and sunny places. They need little water.
...	...	....	...	...
n	Agave	...	Succulent	Plants leaving in dry, hot and sunny places. They need little water.

In this case a lot of data are redundant and, moreover, such solution would increase the likelihood of typing errors during the data input phase. It is much more comfortable breaking data in two separate tables, respectively CATEGORIES (the father tables) and PRODUCTS (the child one). By doing so, all information pertaining a specific category are typed at first and only once. Next, for each product, all relevant data are inserted, and a specific category is selected. Owing to these issues, the logic diagram representing the relationship between the tables is the one shown by fig. 1.2.

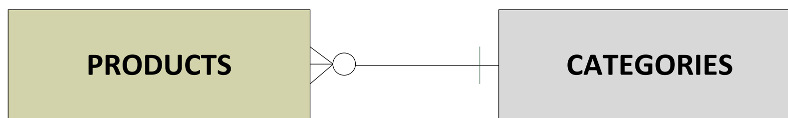

Fig. 1.2. *Logic scheme of an OTM relation*

Now the problem is «how to implement this conceptual scheme». The solution is straightforward; everything is based on the use of the PKs. We anticipate that, alone, the PKs are not sufficient to define a OTM relation and they have to be coupled with a second type of referential keys, namely *the* ***Forward Keys (FKs)***.

Let us consider the previous example once again and let us suppose that the PRODUCTS and CATEGORIES tables are defined as in Tables 1.4 (a) and (b).

Tab. 1.4 (a)
PRODUCTS

Product_ID (PK)	Name	...	On Hand	Reorder Level
1	Aloe		100	50
2	Mouse Tail		80	25
...	...	....	...	...

Tab. 1.4 (b)
CATEGORIES

Category_ID (PK)	Name	Description
1	Succulent	Plants leaving in dry, hot and sunny places. They need little water.
2	Flowers	Cut flowers without roots to be hold in pots
...	...	...

Both tables have a primary key (PK) and, although it is conceptually obvious that they are have a OTM relationship (i.e., many plants belong to the same category), this relation is not obvious from the way in which the two tables are defined. For instance, we already know (see Table 1.3) that both the Aloe the Rat Tail are succulents plant, yet we cannot trace this information by analysing data contained in both table**. To make the relation clear we need to add an additional field to each record of the PRODUCT table that, as known, is the "son" table of the OTM relation**. More specifically, it is exactly this additional field (i.e., the FK) that specifies the category to which each product belongs to. Also, and more important, since we are creating an OTM relation, *the FK in the PRODUCT table can be repeated more than once* (all the products belonging to the same category will have the same FK), *yet for each different value of the FK in the PRODUCT table, there must be a single corresponding value of the PK in the CATEGORY table*.

Obviously, for a relationship to be of OTM type it is necessary that, *for all values of the field that identifies product category (in the table PRODUCTS) there must be one and only one equivalent field in the PRODUCTS table*. This is shown in Table 1.5.

The use of a FK makes it easy to understand that Aloe and Mouse Tail are succulent plants, while roses are flowers*. Obviously, if one or more values of the FK (in the PRODUCTS table) did not have a correspondent value in the PK of the CATEGORIES table, this would generate a referential integrity error,* as it would imply that a product belongs to a category that does not exist yet. This integrity check (i.e., the value of the FK of the child table matches the value of the PK of the father table) is automatically performed by the RIC of any commercial RDB.

Tab. 1.5.
Forward Key

Product_ID (PK)	Cat_ID (FK)	Name	…
1	1	Aloe	…
2	1	Mouse Tail	…
…		…	….
m	2	Cut Roses	…
…	…	…	…

Summarizing we can say that, to create a OTM relation between two table A and B, with B being the father table:

- ***Both tables must have a primary key*** PK_A ***e*** PK_B***;***
- ***The Son Table A must have a forward key*** $FK_{(A \to B)}$ ***creating the link with the Father Table B;***
- ***All values assigned to the forward key*** $FK_{(A \to B)}$ ***must belong to the same domain (i.e., be a sub set) of the value of the primary key*** PK_B***, i.e.,*** $FK_{A \to B} \subseteq PK_B$.

It is important to note that <u>*the label given to the FK can be different from the label of the corresponding PK*</u>. For instance, in Table 1.5 the FK is labelled as Cat_IC, whereas the corresponding PK is labelled as Category_ID. Also, three other important differences between a PK and a FK are the following ones:

- ***The PK must be unique, the FK can be non-unique (most of the times it is not unique);***
- ***There may be some values of the PK (of the father table) that have no correspondence in the values of the FK (of the son table); each value of the FK (of the son table) must have a correspondence in the PK (of the father table)***
- ***The PK cannot be null[1], the FK can be null.***

The last property is particularly important. <u>**The PK cannot be null**</u>, as it is needed to identify (and make unique) a record. Conversely the <u>**FK can be null**</u> <u>*because there may be some (rare) case of "sons without fathers".*</u> Let us consider the following example. The plant shop that we are considering has never sell carnivorous plant, so this kind of plants does not appear in the CATEGORIES, yet. Now, let us suppose that the shop decides to add to its product catalogue a 'venus flytrap' (a common carnivorous plant). In this case there is the need: *(i)* to add the new category, *(ii)* to add the new product and *(iii)* to crate the linkage, by assigning to the FK of the 'venus flytrap' the same value of the PK (let us say 23) assigned to the new 'carnivorous plant' category. Alternatively, it is also

[1] *A null value is a blank value in a field of a record. A null value is not zero. Zero is a number, null is a missing value. The concepts are very different.*

possible to add the new product, without creating the new category. To this aim it is sufficient to add the new record leaving blank (i.e., null) the FK field.

OTO Relations

<u>OTO relations are a specific type of OTM relations used to split massive tables</u> (massive in terms of number of fields and of occupied space) in sub tables, so as to reduce the total amount of occupied memory and, most of all, to speed up the operations (i.e., SQL queries) performed on the RDB.

Let us suppose that A is the son table (in this case it is better to talk in terms of **<u>derived table</u>**) and that B is the father table (or better the **<u>original table</u>**). In case of an OTO relation between A and B, each record of A must be linked with a single record of B, but not all the records of B must be linked with a record of A. In other words, the link is of a *"one to one"* type, but the relation is not bidirectional, as some records of the original table B could not have a correspondent record in the derived table A.

More formally, **<u>in an OTO relation, the link can be defined by operating, directly, on the primary keys PK_A</u>** and PK_B of the derived and of the original table, respectively; that is to say that the primary key PK_A also acts as the forward key FK_A of the OTO relation among A and B[2].

To clarify this concept let us consider two tables COMPONENTS and DRAWINGS; each component (or part type) may have a specific 2D and 3D drawing and, clearly, each 2D and 3D drawing corresponds to a specific component. So, the relation among COMPONENTS and DRAWINGS is evidently of the form OTO. It is also clear that COMPONENT is the original table, since drawings belong to a component and not vice versa. Thus, the PK of the COMPONENTS table will be used also as FK to create the link with the original table. This is shown by Fig. 1.4 ad in Tables 1.6 (a) and (b).

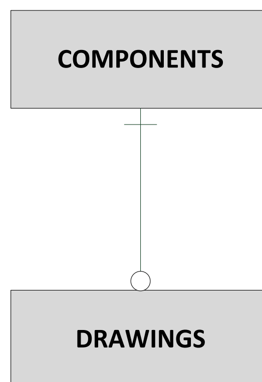

Fig. 1.4. *An OTO relation*

[2] *Actually, it would be possible to add a specific field to be used as FK, but this is not strictly necessary.*

Tab. 1.6. (a)
Original Table

Comp_ID	Component	Seller	Price
1	Pump	Smith	100$
2	Gear	Smith	5$
3	Compressor	Robertson	140$

Tab. 1.6. (b)
Derived Table

Comp_ID	2D Drawing	3D Drawing
1	Pump2d.dwg	Null
3	Compressor2d.dwg	Compressor3d.dwg

In this case we have three components, whose basic features are described by the fields of Table 1.6 (a). Components may have 2D and 3D drawings, that are stored (as files) in the fields of the records of Table 1.6 (b), which is in relation OTO with table 1.6 (a). Specifically, since the link is made on the primary key (i.e., Comp_ID), from the analysis of the tables it is immediate to see that the pump has only a 2D drawing, that the compressor has both a 2D and a 3D drawing, whereas the gear has neither a 2D nor a 3D drawings. Also note that, also the following solution is acceptable:

Tab. 1.6. (c)
Derived Table

ID (PK)	Comp_ID (FK)	2D Drawing	3D Drawing
1	1	Pump2d.dwg	Null
2	3	Compressor2d.dwg	Compressor3d.dwg

In this case an additional column (ID) has been added and used a primary key, whereas Comp_ID is used as forward key.

Anyhow, both solutions make clear the way in which an OTO relation is built. Yet there is another fundamental question that still need to be answered: *«why do we need to use a OTO relation when all the information could be included, without duplication, in a single table?»*. As a matter of fact, one could argue that a table as the one of Tab. 1.7 is perfectly licit: data are logically combined without any redundancy and/or duplications.

Tab. 1.7.
The combined table

ID	Component	Seller	2D Drawing	3D Drawing
1	Pump	Smith	Pump2d.dwg	Null
2	Gear	Smith	Null	Null
3	Compressor	Robertson	Compressor2d.dwg	Compressor3d.dwg

The reason why it is wise to split this table into two sub tables is quite simple. Note that, apart from the third record, all the others have some null values. Generally speaking this is not a big problem, null fields are allowed, and their use is rather frequent. So where is the problem? The problem link to the fact *that a null value occupies as many MB of memory as the ones occupied by data type associated to the corresponding field*. Let us say that we have a record made of two fields, both numerical, declared as Integer. In this case, since in a 64 bit systems an Integer value occupies 8 bytes, each record will occupy 16 bytes, even if some fields are left blank (i.e, null). In other words, as soon as a record is created, an amount of memory equal to the maximum number of MB required to store all the data of the record is immediately reserved.

This problem is negligible if a field contains number or string (these data types necessitate just a couple of bytes), but must be considered if a field contains a heavy file (pdf, dwg, doc, xlt, dwg, etc) or a picture (jpg, tif, etc.). For instance, in the case of Tab. 1.6, if we assigned a maximum space of 50 MB for a 2D dwg file and a maximum of 150 MB for a 3D dwg file, then a null value in the 2D and in the 3D drawing column would occupies 50 and 150 MB, respectively. Thus, *to reduce memory usage, in case of records with very heavy fields, it may be useful to split the records in two or more parts, using OTO relations:*

- *The main table contains all 'light' data;*
- *The derived table contains all the 'heavy' data*.

We also note that this strategy is particularly advantageous if heavy data are rarely consulted and if not all the records (of the original table) use them. Indeed:

- *If the heavy data are rarely consulted then, most of the times SQL queries will operate only on the original table (the COMPONENTS one in our example); with a clear advantage in terms of time;*
- *If some records,* such as the gear that has neither 2D nor 3D drawings, *do not need any of the heavy fields, then by splitting the table the occupied memory can be highly reduced, as the derived table is more compact (it has less rows than the original one) and the number of null values is very low.*

4. Self Referencing Relationship (SRR)

SRRs are **<u>used to represent hierarchical structures</u>** such as organizational charts, functional decompositions, bills of materials, etc. In this case, as sketched by Fig. 1.5, a relation is created among the records belonging to the same table. To this aim, the primary key is replicated (in an additional field) and used as the forward key of traditional OTM relation.

For example, making reference to our Database, you can use an SRR to represent the organization of the management structure, as shown in Figure 1.6.

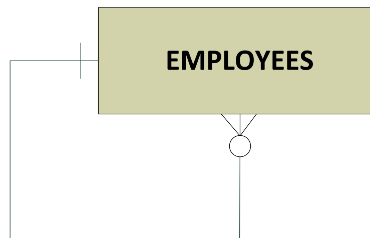

Fig. 2.5. *A SRR for the creation of an organizational chart*

To better explain how a SRR is created, let us consider the following organizational chart:

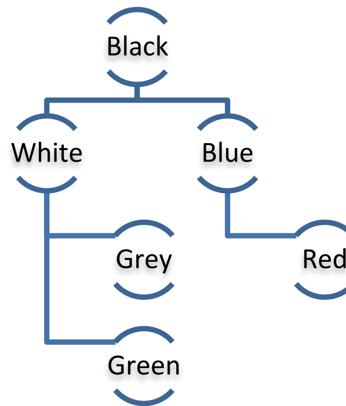

Fig. 1.6. *A simple organizational chart*

To recreate the hierarchical structure defined by the organizational chart of Fig. 1.6 a table as the one of Tab. 1.8 can be used. As it can be seen, the field ID_Employee is used as PK and the field Respond_To is used as FK. It is thus evident that white and blue respond, directly, with the boss and they are responsible (of the activities) of Grey and Gren and of Red, respectively. Also note that, since in the third record (the one of Black) the FK is null, Black does not respond to anybody; so, he must be at the vertex of the hierarchy and, indeed, he is the boss.

We terminate this section by noting that, *<u>anytime a SRR is used, particular attention must be paid to define accurate standards to ensure referential integrity</u>*. In particular, one has to properly code all the **<u>rules that</u>**, while

entering or editing data, **are needed to prevent the occurrence of dangerous circular references** that could lead to an infinite loop. For instance, assume that the employee A is the direct superior of the employee B who, in turn, is the direct superior of employee C. It is evident that A must be higher (two level above) than C in the hierarchy. Vice versa, if an input error was made and C was indicated as the direct superior of A (i.e., $FK_C = PK_A$) this would generate the circular reference A⟶ B⟶ C⟶ A. Thus, running a query to reconstruct the hierarchy, the SRR relation could not be closed and this would lead the application into an infinite loop, effectively blocking the execution.

Tab. 1.8.
The corresponding SRR

ID_Employee (PK)	Surname	Job	…	Respond_To (FK)
1	White	Nurseryman	…	3
2	Blue	Warehouseman	…	3
3	Black	Owner and Clark	…	Null
4	Red	Warehouseman	…	2
5	Grey	Nurseryman	…	1
6	Green	Nurseryman	…	1

5. Many-To-Many Relationship (MTM)

Let us consider the tables CUSTOMERS, ORDERS and PRODUCUTS shown by the *entities-relations* diagram of Fig.1.1. Evidently, a customer can make more than an order and, conversely, an order is relative to a specific customer. Thus, the relation among COSTOMERS and ORDERS must be of the form OTM (i.e., CUSTOMERS is the father table, ORDERS is the son one and a Customers_ID field will be used as an additional field of the ORDERS table, so as to create the join element or forward key). Up to now there is nothing new. What about the relation between ORDERS and PRODUCTS? In this case the situation is rather different, indeed: an order can contain more products (a set of products ordered altogether) and, of course, the same product can be ordered more than once either in the same order or in different order. So, in this case the relation is of the Many to Many form.

How can we implement this relation in a database? A first attempt could be that to codify this relationship by introducing a foreign key both the PRODUCTS and in the ORDERS table. Unfortunately, this is practically impossible: since an order may correspond to many products and a product may correspond to many different orders, it is impossible to associate a unique foreign key to each product and, similarly, it is impossible to assign a unique foreign key to each stock.

The only possible solution is that **to decompose a MTM relation in two distinct OTM relations. How to do so? Fortunately, the process is straightforward.**

Intuitively, we can make the following consideration. Given *p* products and *o* orders, the maximum number of products-orders couples equals $p \times o$. For example, if $p = 5$ and $o = 2$, the possible 10 couples are as follows (see Table 1.9 (a)):

Tab. 1.9 (a).
The full set of the possible couples

Products ID	Orders ID
1	1
1	2
2	1
2	2
3	1
3	2
4	1
4	2

Clearly, not all the $m \times n$ couples obtained in this way may have a physical sense, for instance if Order 1 was made by product 2 and 3, the couples 1-1 and 1-4 (highlighted in grey in the table) does not have any meaning. Anyhow, this is not a problem, it is sufficient to get rid of all the meaningless couple and the linkage between products and order is fully reconstructed. More formally, what we have just done is a “cartesian product” of all the values taken by the PK of the PRODUCT Table, with all the values taken by the PK of the ORDERS Table. Note that we used the PKs because they are the only fields that certainly refer to a specific record of a table. Also, making the Cartesian Products, we implicitly created a relationship between two atomic data (i.e., the two PKs). Actually, by doing so we have created a new meta-data (i.e., a new table) expressing the relationship between the original tables PRODUCTS and ORDERS.

Let us now consider in more details the fields of this newly generated tables: they both are made with values belonging to the Domain of the PK of other tables (in this case the PRODUCTS and ORDERS tables respectively). So, these fields have all the properties to be the FKs of a OTM relationship. Also, since the FK belongs to the “Son” table, the newly created table must be, at the same time, son of the PRODUCTS and of the ORDERS table. That’s it we have intuitively decomposed a MTM relationship into two OTM relationship.

Owing to what we’ve just said, to decompose a MTM relationship in a OTM relationship it **is sufficient to add a new table** (i.e., ORDERS_DETAILS in the case of Fig. 1.1), which is typically referred as a ***bridge table*****, that has two foreign keys linked to the primary keys of the original table that are in MTM relation**. Table 1.9 (b) shows an example of what said above:

Tab. 1.9 (b).
A first example of a bridge table

ORDERS DETAILS	
Order_Id	**Product_Id**
1	1
1	2
1	3
2	1
2	4
…	…
N	1
N	9

Obviously, Order_Id is a FK related to the PK of the ORDERS table and, similarly, Product_ID is a FK related to the PK of the PRODUCTS table. Thus, from the ORDERS DETAILS table it is easy to see that, so far, a total of *n* orders have been issued. It is also easy to see that the first order contains product #1, product #2 and product #3; similarly, the last order contains product #1 and product #2. It is also evident that the relation between ORDERS and PRODUCTS is of the form MTM because neither the FK Order_ID nor the FK Product_ID is unique.

Also note that, in this form, although Tab. 1.9 crates the linkage between PRODUCTS and ORDERS, some crucial data are still missing. For instance, we know that the first order contained product #1, but we know neither the ordered quantity nor the purchase price and, also, we know neither the issuing date nor the due date. Where can we introduce these data? To answer this question, *anytime we have a bridge table, we have to clarify whether or not the value taken by a certain data depends on the combination of two fields of the tables that are in MTM relations.* Specifically, we have that:

- It is easy to understand that both the issuing date and the due date are certainly linked to an order, but they do not depend on the products contained in the order. In other words, all the products contained in an order have been ordered the same day and are expected to be delivered the same day.
- Conversely, both the quantity and the purchase price (the purchase price depends on the order quantity but also on the customer who issued the order, who could benefit of special offers and/or discounts) depend on the order and also on the ordered price. In other words, each product of an order could have a specific price and ordered quantity.

Due to the above-mentioned reasons, it is thus evident that the issuing date and the due date will be fields of the ORDERS table; conversely, ordered quantity and price will be fields of the bridge table, as shown in Tab. 1.10.

Tab. 1.10.
A second example of a bridge table

ORDERS DETAILS			
Order_Id	**Product_Id**	**Ordered Quantity**	**Purchase Price**
1	1	5	1000€
1	2	10	2000€
1	3	1	100€
2	1	1	140€
2	4	24	15000€
…	…		
N	1	15	1000€
N	9	15	800€

Obviously, the information concerning the ordered products can be found, immediately on the PRODUCTS table that, as we have said, is now in OTM relation with the ORDERS_DETAILS table. For instance, product #1 is a succulent plant called Aloe. Similarly, all the information related to the customer who issued an order can be found on the CUSTOMERS table that, as known, is in OTM relation with the ORDERS table that, in turn is in OTM relation with the ORDER_DERAILS table (see Fig.1.1. for more detail). Thus, going along all these relationships it is possible to connect a customer with an ordered product. In other words, we could say that the relational structure here proposed accurately represent the purchasing process followed by a customer who wants to buy an item.

The overall relation is now clear, but is there anything missing in Tab. 1.10? We know that, as a general rule, each record of a table must be unique and that, in order to fulfill this requirement, a PK is needed. At a first sight we could argue that Tab. 1.10 is wrong because, at least apparently, a PK is missing. However, at a closer look, we can see that, although a field specifically designated as PK (i.e., an ID) does not exist, all the records of Tab. 1.10 are different. Also, this is not only a lucky fluke, but it is a property that descends from the way in which Tab.1.10 has been created (i.e., taking a subset of the 2-tuples obtained making the cartesian products of unique values) Indeed, the same couple Order_Id and Product_Id cannot and must not appear more than once.

Let us consider the records of Tab. 1.11 (a):

Tab. 1.11. (a)
An integrity constraint error

ORDERS DETAILS			
Order_Id	**Product_Id**	**Ordered Quantity**	**Purchase Price**
1	1	5	1000€
…	…	…	
1	1	2	400€

These records are wrong, as they refer to the same order and to the same ordered item. Thus, the information is redundant and the records should be aggregated as in Tab. 1.11 (b):

Tab. 1.11. (b)
An integrity constraint error

ORDERS DETAILS			
Order_Id	**Product_Id**	**Ordered Quantity**	**Purchase Price**
1	1	7	1400€
…	…	…	

So, if we jointly consider the values taken by Order_Id and Product_Id (in the same record) we certainly obtain a unique value. So, Order_Id and Product_Id can be used, jointly, to define the PK of the ORDERS_DETAILS table. Also, the introduction of a specific ID field (to be used as PK) would be licit, but the use of Order_Id together with Product_Id is a better choice. Indeed, it avoids the addition of an extra field and, most of all, it automatically prevent integrity constraints errors as the one of Tab. 1.11 (b).

All the above mentioned issues are generic and apply to any bridge table. So, summarizing, we can say that, in **case of bridge tables, it is wise to use the FKs (taken conjointly) also as PK. This is one of the rare circumstances in which a PK based on two fields is preferable than a PK based on a specific and single field**.

Lastly, we observe that **it is possible to create a document - in this case a Purchase Order - for each record of the ORDERS table**, using, as document's rows, the records of the bridge table having the same Order_ID value. An example is given by Fig. 1.5.

Order ID #1 Issue Date 01/01/2016
Customer - Alfa Company
Due Date 315/01/2016

Product ID	*Product Name*	*Quantity*	*Price*
#1	Alone	5	1000€
#2	Mouse Tail	10	2000€
#3	Fertilizer	1	100€

Fig 1.5. *A PO document*

5.1 The special case of a Bridge Table with a PK formed by three fields

Please note that, *in some (rare circumstances), using the two FKs may not be enough to define the PK*[3] *of a bridge table,* and a third field should be used to the scope. Consider for example, the Relational Data Base of a Public Library made by the following main Tables: *(i)* Authors, *(ii)* Titles, *(iii)* Books, *(iv)* Loans and *(v)* Customers. Please note that with "Titles" we refer to a book in general, whereas with "Books" we refer to the physical copies available in the library i.e., a Title could be "La divina commedia" by Dante, whereas a Book could be "La divina commedia" in hard cover edited by "Casa editrice Fiorentina". Also note that, as can be seen in Figure 1.6:

- There is a OTM relationship between Authors and Titles and between Titles and Books;
- There is a MTM relationship between Books and Customers (i.e., a customer may have made several loans and, similarly, the same book may have been loaned by different customers);
- The "Loans" table is the bridge table splitting the above mentioned MTM relationship (between Books and Customers) into two OTM relationships.

[3] A bridge table containing the books borrowed by the customers of a public library is a good example. Can tou figure it out, why, in this case Customer_ID and Book_ID are not sufficient to form a Primary Key?

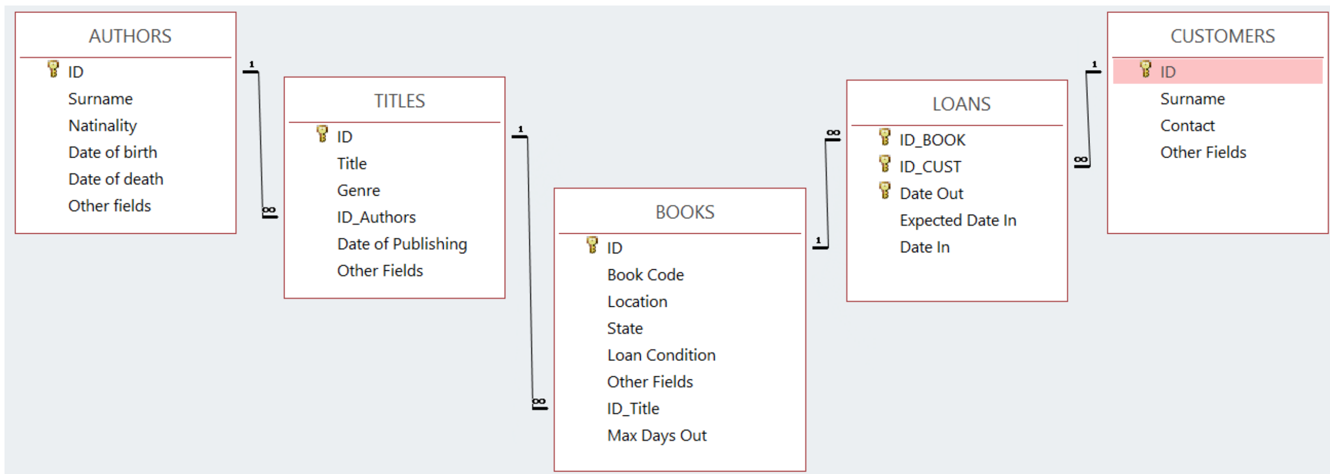

Fig 1.6. *The RDB of a Library*

What is important to note is the fact that, the PK of the LOANS table is based on three distinct fields: ID_Book, ID_Cust and Date Out, respectively. Whereas the first two fields are the FKs used to link this table with the CUSTOMERS and the BOOKS ones, Date Out is a field containing the data of the loan. Can you figure it out the reason why this additional field is needed, to obtain a unique key?

The answer is straightforward: contrary to the warehouses' case, now the FKs alone are not enough to assure the unicity of the primary key. Indeed, since a customer may load the same copy of a book more than once, this rare but plausible case, would generate a duplicate of a primary keys formed by the junction of the two FKs. To better clarity this concept, let us suppose that Customer ID = 10 borrowed Book ID = 20 on the 17th of December 2000 and that he or she returned the book on the 15th of January 2001. The record corresponding to these two transactions (loan and return) would be as follows:

ID_Book	ID_Customer	Date_Out	Date_In
20	10	17/12/2000	15/01/2001

Next, if the customer should borrow that book once again, let us say on the first of September 2004, the registration of the loan would be the following one:

ID_Book	ID_Customer	Date_Out	Date_In
20	10	17/12/2000	15/01/2001
20	10	01/09/2004	Null

As it can be seen, ID_Book and ID_Customer take the same value in both records; yet, what differentiates these records is the date of the loan (i.e., the same book cannot be loaned by the same customer on the same date) that, consequently, can be used as a third field to create the PK. Also note that choosing Date_In as the third

fields, would have be wrong, as this fields remains "Null" till the book is returned by the customer. For this reason Date_In cannot be used as part of the PK (i.e., it does not respect all the requirements of a PK).

6. Many-To-Many Relations and Self Referencing Relationships

Sometimes *complex hierarchical structures must be defined and, to this aim, MTM and SRR relationships may be used conjointly*. To clarify this concept let us consider a Bill of Material (BOM). As known, a bill of materials is the list of raw materials, sub-assemblies, intermediate assemblies, sub-components, parts etc., that are needed to manufacture an end product. Furthermore, the BOM also specifies both the quantities and the processing time (i.e., Lead Time) of each material, assembly and component.

It is evident that an end product needs several components, several parts and several row materials; similarly, the same component or raw material can be required (maybe with different quantities) by many end products. Parts and components standardization (and sharing among end products) is, indeed, a wise way to reduce variability and to increase production flexibility. Thus, the relation among END PRODUCTS and COMPONENTS (without loss of generality we can assume that parts, components, sub-assemblies and raw materials can be included in the same table) is definitely of the MTM type. Thus, a bridge table, which we could call BOMS (and that will use End_Product_Id and Component_Id both as primary and forward keys) is needed to split the MTM relation into two distinct OTM relations. It is also easy to see that, to fully define a BOM it is not sufficient to specify which components are used by a certain end product, but we also have to define: *(i)* needed quantities, *(ii)* processing/machining/delivering time (i.e., Lead Time) and, above all, (iii) the level of the hierarchy occupied by each component. To this aim *Quantity* and *LT* fields must be include in the BOMS table too and, to define the levels of the hierarchy, also a SRR must be defined on the same table.

What we have just said is graphically shown by Fig. 1.7.

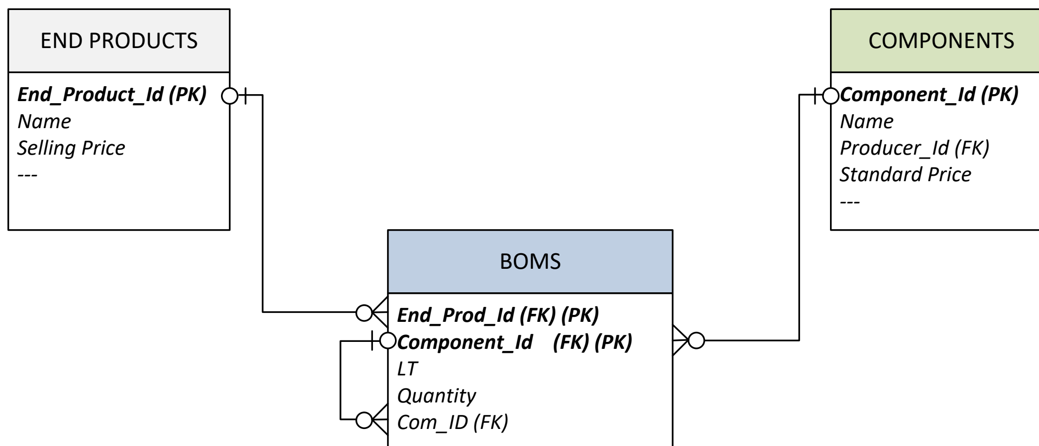

Fig 1.7. *Creating a BOM using MTM and SRR relationships*

In this way, to re-create the BOM of a specific end product, let say the *n*-th one, it is sufficient to find all the records of the BOMS table where the End_Prod_Id field is equal to the PK of the *n*-th product and, next, to use the information coded by the SRR to recreate the hierarchy.

This may sound a little bit confounding, so an example could help to understand it. Let us consider the EPA End Product (shown in Figure 1.8).

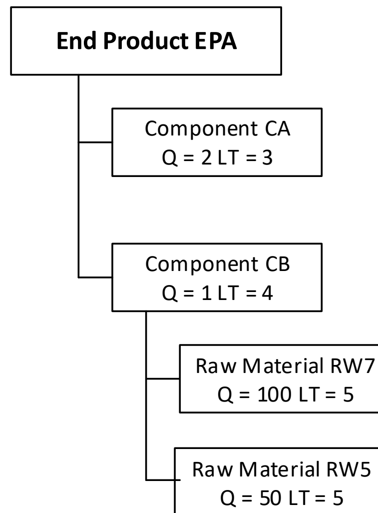

Fig. 1.8. *A simplified BOM*

As it can be seen, this end product only requires two first level components (i.e., CA and CB), the latter one being obtained from raw material RW7 and RW5. Thus, in total EPA requires 4 items: two first level components and two raw materials. Consequently, its BOM will be coded with 4 records, as shown in Table 1.12(a), where only the records related to EPA are highlighted in dark blue.

Tab. 1.12 (a).
The BOMS Bridge Table

BOM				
End_Prod_Id (FK)	Comp_Id (FK)	Quantity	LT	Com_Id (SRR)
EPA	**CA**	**2**	**3**	**Null**
EPB	CC	1	1	Null
EPC	RW1	10	5	CC
EPA	**RW5**	**50**	**5**	**CB**
EPA	**CB**	**1**	**4**	**Null**
…	…	…	…	…
EPA	**RW7**	**100**	**5**	**CB**

Note that End_Prod_Id and Comp_Id are the two FKs (that also generated the unique PK), whereas Com_Id is the field used to create the SRR. In this regard, it is worth noting that only the first level components (i.e., CA and CB) have a Null value in the Com_Id field. This is because, in the hierarchy, above them there is EPA that, being an end product, is not included among the components. In other words, both CA and CB are top level components. Conversely, for both raw materials RW5 and RW7 the Com_Id field is equal to CB, since they are both required to generate the top-level component CB. In this way the whole hierarchical structure of the BOM is correctly reproduced.

Nonetheless, to be frank this solution, although frequently used, does not assure high performance, as it implies data redundancies. To explain this concept let as consider a second end product (let it be EPN) requiring the same top-level component CB used by the end product EPA, of figure 1.8. Due to this new entry, if EPN requires a total amount of 2 CB, the bridge table turns into the following one:

Tab. 1.12 (b).

Addition of a new end product sharing the same top-level component CB

BOM				
End_Prod_Id (FK)	**Comp_Id (FK)**	**Quantity**	**LT**	**Com_Id (SRR)**
EPA	**CA**	**2**	**3**	**Null**
EPA	**CB**	**1**	**4**	**Null**
EPA	**RW5**	**50**	**5**	**CB**
EPA	**RW7**	**100**	**5**	**CB**
...	...	...	...	...
EPN	**CB**	**2**	**4**	**Null**
EPN	**RW5**	**50**	**5**	**CB**
EPN	**RW7**	**100**	**5**	**CB**

As it can be seen, apart from the value of the first field (i.e., End_Product), the records highlighted in dark and light blue are identical, as they both codify the structure of the same top-level component, that is CB.

This critical issue occurs any time a component, which is internally produced (through the assembly of parts and/or the machining of raw materials), is shared by different end products.

To obviate this problem, the structure of Figure 1.7. must be slightly modified. But how? To give a proper answer to this tricky question, let us consider the following two facts:

- Internally made components are obtained starting from basic components and/or raw materials; as such, as for the end products, they should be placed in the END PRODUCTS table, too;
- Internally made components are part of the end products manufactured by the company; as such, as for raw materials and low-level components, they should be placed in the COMPONENTS table, too.

Owing to these considerations we get to the conclusion that components should be placed in two different tables at the same time… This is, evidently, a non-sense: we cannot count on ubiquity and, even if so, this would definitely lead to data redundancy.

So, what? Well, up to this point we get to the conclusion that internally manufactured components should be placed into two different tables. Since this is impossible, we need to reconsider the situation from a different point of view: rather than operating on the end products we need to operate, directly, on the END PRODUCTS and COMPONENTS tables. To ensure that components belong to both tables, we need to join or better, to collapse the END PRODUCTS and the COMPONENTS tables in a single and more generic PRODUCTS table, containing all the end products, components and raw materials.

Now a second question arises. When we collapse the original table in a single one, what does it happen to the original OTM relationships? Before we collapsed END PRODUCTS and COMPONENTS in a single table there were two OTM relationships connecting END PRODUCTS and BOMS and COMPONENTS and BOMS, respectively. These relationships must be preserved because, even if the original tables have collapsed into a single one, there is still a MTM relation between parts and components. So, in the new solution there must be an OTM relation going from PRODUCTS to BOMS and there must be a second OTM relation going backward from BOMS to PRODUCTS. This double-linked OTM relationship makes it clear that the “End_Prod_Id” and the “Comp_Id” fields of the BOMS table are two forward keys linked to the same primary key of the PRODUCTS table. Also, since the BOMS table and the double-linked OTM relations explicate a relationship among records belonging to the same table (i.e., among end products, components and raw materials of the PRODUCTS table), they also encompass the SRR relation, which is now superfluous. This is graphically shown by figures 1.9 (a) and (b).

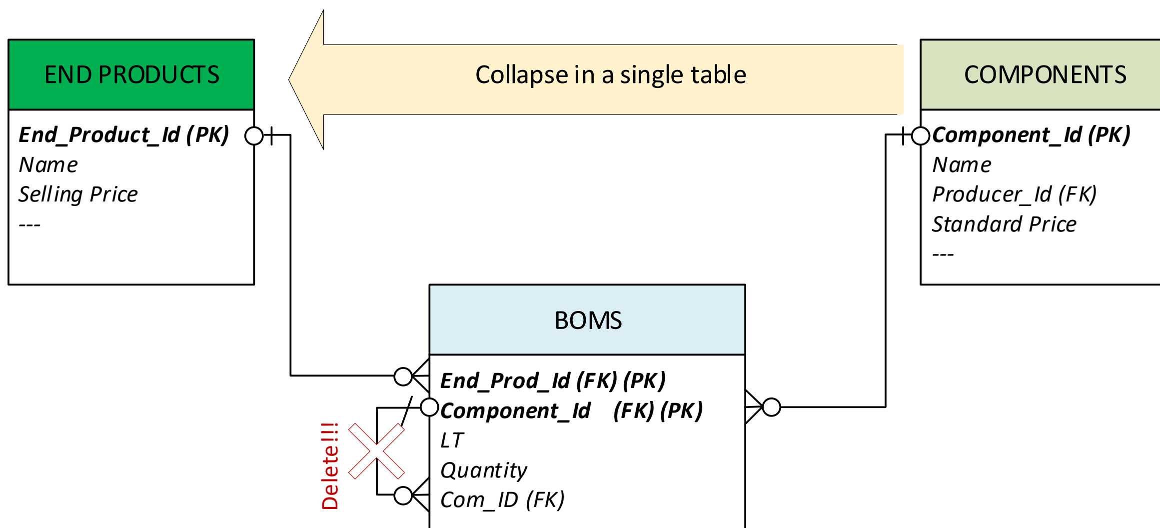

Fig. 1.9 (a). *Modifications to the original scheme*

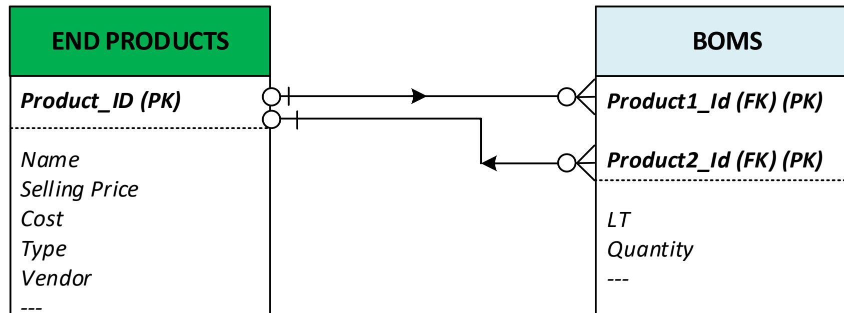

Fig. 1.9 (b). *The new scheme*

Note that the Com_Id field (i.e., the FK of the SRR made on the bridge table), that was originally used to identify the component/product placed at the next higher level of the hierarchy, has been removed from the BOMS table. Consequently, in the new scheme, both the FKs (i.e., Product1_Id and Product2_Id) are linked with the PK (Product_ID) of the PRODUCTS table, and they have the following meaning:

- Product1_Id specifies the product/component that is at the next higher level of the hierarchy;
- Product2_Id specifies the product/component that is at the next lower level of the hierarchy.

In other words, the first link (the one going from PRODUCTS to BOMS) specifies which product/component is at the higher level of the hierarchy, while the other one (going backward from BOMS to PRODUCTS) specifies which product/component is at the lower level of the hierarchy. For instance, if there was a record with Product1_Id = PA and Product2_Id = CA, this record would imply that PA uses CA (i.e., PA is at the nest higher level of the BOM, relatively to CA).

To clarify these concepts let us see how the EPA and EPN end products could be represented in the PRODUCTS and BOMS table.

Table 1.13 (a)
PRODUCT Table

Product_ID	Name	Type	Cost	...
EPA	Product_A	End Product	$ 2000	
EPB	Product_B	End Product	$ 1500	
CA	Component_A	Internal Component	$ 200	
CB	Component_B	Internal Component	$ 150	
RW7	Material #7	Raw_Material	$ 28	
RW5	Material #5	Raw_Material	$25	

Table 1.13 (b)
BOMS Table

Product1_ID	Product2_ID	Q	LT
EPA	CA	2	3
EPA	CB	1	4
EPN	CB	2	4
CB	RW5	50	5
CB	RW7	100	5

As it can be seen, all the end-products, component and raw materials are included in the PRODUCTS table; conversely only products and components that are not at the end (lower level) of the hierarchical structure are included, as FKs, in the BOMS table. Conversely, components and/or raw materials that are leaves of the hierarchical structure (i.e., bottom level elements such as CA, RW5 and RW7) are not included in the BOMS table. Also note that the records of the BOMS table only detail the first level of the BOM of each product (or component) coded in the first field (i.e. Component_1) of the table. For instance:

- The first two records detail the first level of the BOM of Product A;
- The third one the first level of the BOM of Product N;
- The fourth and fifth records detail the first level of the BOM of the of component B, that is part of both the first level of the BOM of both Product A and Product B.

In order to recreate the whole hierarchy a recursive approach is needed:

- An end product is selected and used to filter records of the BOMS table.
- The components and/or Raw materials listed in the second field (i.e. Components_2) of the remaining records of the filtered table, correspond to the first level of the BOM of the selected end product;
- All the above-mentioned components are used, one at a time, to filter the BOMS table;
- The above-mentioned steps are repeated to form the whole hierarchy.

For instance, using the end-product EPA to filter the BOMS table, one would find components CB and CA. This means that CA and CB are the first level of the BOM of EPA.

Next, using CA to filter the table, one would not find any other components or raw materials. Thus, CA is a leaf of the hierarchy. Conversely, filtering using CB would return two raw materials, RM5 and RM7. Hence, product EPA is made by CB (first level), which is made (second level) by RW5 and by RW7. Lastly,

filtering using either RW5 or RW7 would not return any additional raw material. So RW5 and RW7 are leaf of the hierarchy and the BOM of Product EPA is completed.

6.1 Reconstructing the BOM through recursion

For the sake of completeness, a VBA code implementing a simple recursive procedure that, starting from a BOMS table, generates and print the BOM, is reported below.

Briefly, the procedure exploits the use of Jagged Array, a data structure that makes it possible to nest, one in another, arrays of different length. A jagged array is a one-dimensional array whose elements are themselves one-dimensional arrays, so, in practice, it is a data structure composed of a parent array that contains, within it, a series of child arrays. At first glance, one could argue that a jagged array coincides, exactly, with a two-dimensional array. This idea, although conceptually correct, is not entirely true. What differentiates a two-dimensional array from a jagged array is the fact that: *(i)* while the rows of a matrix must necessarily contain a same number of elements of the same type, *(ii)* the child array of a jagged array can have different dimensions and can hold different data types.

A graphical display of what we’ve said above is shown in the following diagram that compares a traditional matrix **M**, with 3 rows and 3 columns and a jagged array **J** with 3 rows and variable number of columns.

	A(1,1)	A(1,2)	A(1,3)
A	A(2,1)	A(2,2)	A(2,3)
	A(3,1)	A(3,2)	A(3,3)

	J (1) (1)	J (1) (2)	
J	J (2) (1)		
	J (3) (1)	J (3) (2)	J (3) (3)

To create a jagged array in VBA, the “parent” array must be declared as variant; all the “child” vectors, may be of any kind, but most of the times, it is useful to declare as variant the child vectors, too. For example, suppose you want to create a jagged array like the one in the previous figure and suppose that the child vectors are of Integer, String and Double type, respectively. A possible implementation is shown below, where it is also shown, how to access to the value of a jagged array with the n-index notation J(i)(i)(k)…(z):

```
Dim J(1 To 3) As Variant
Dim j_1(1 To 2) As Integer
Dim j_2(1 To 1) As String
Dim j_3(1 to 3) As Double
'Generation of the integer vector j_1
j_1(1) = 1
j_1(2) = 2
'Generation of the string vector j_2 (actually a scalar)
j_2(1) = "a"
'Generation of the double vector j_3
j_3(1) = 0.1
j_3(2) = 0.2
j_3(3) = 0.3
'Generation of the jagged array J
J(1) = j_1
J(2) = j_2
J(3) = j_3

'Data access
Dim I As integer
Dim S As String
Dim D As Double
I = J(1)(2) 'Second element of the vector contained in the first element of the parent array → return 2
S = J(2)(1) 'First (and only) element of the vector contained in the second element of the parent array → return "a"
D = J(3)(3) 'Third element of the vector contained in the third element of the parent array → return 0.3
J(3)(2) = J(3)(2) + 0.05  'It adds 0.05 to the original value of 0.2
```

Specifically, to recreate the BOM of a product with *L* levels:

- *L* variant arrays $L_1, L_2, \ldots, L_L$ will be used; L_1 represents the first level of the BOM, L_2 represents the second level of the BOM and so on, up to the last level codified by vector L_L;
- Each generic array L_j will be made of $(2 \times N)$ elements, with *N* equal to the number of components of the *j*-th level of the BOM;
- Each element $L_j[i]$ in 'odd' position $i \in \{1, 3, 5, \ldots\}$ will contain the Identification Code (Id) of the $(i \backslash 2)$-th[4] component of the *j*-th level of the BOM, if $i \neq 1$. If i = 1, $L_j[i]$ corresponds to the Id of the first component of the j-th level of the BOM.

[4] Where \ is the integer division sign.

- Each element $L_j[i + 1]$ in pair position *(i+1)* will be Null, if the preceding element $L_j[i]$ corresponds to a leaf of the BOM; otherwise it will contain another array. In the latter case $L_j[i + 1]$ will contain the array $\mathbf{L_{(j+1)}}$ that codifies the next level of the BOM (i.e., all the sub components, or raw materials, of the product coded by the $L_j[i]$).

For instance, the end-product EPA (that we considered before), will be coded as:

- $L_1[1]$ = 'EPA', $L_1[2]$ = $\mathbf{L_2}$
- $L_1[2][1]$ = 'CA', $L_1[2][2]$ = Null, $L_1[2][3]$ = 'CB', $L_1[2][4]$ = $\mathbf{L_3}$
- $L_1[2][4][1]$ = 'RM5', $L_1[2][4][2]$ = Null, $L_1[2][4][3]$ = 'RM7', $L_1[2][4][4]$ = Null

Indeed, EPA has the following three levels BOM (figure 1.10 (a)):

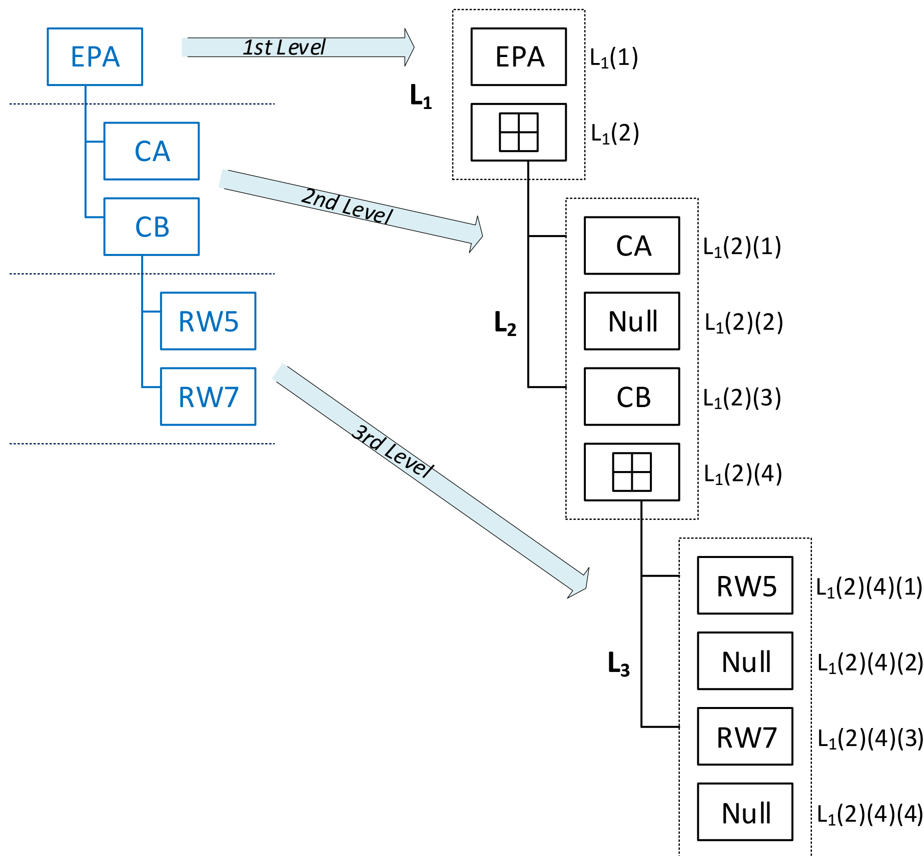

Figure 1.10 (a). *EPA's BOM coded as a jagged array*

This is also shown in the VBA's screen shot of Figure 1.10 (b), where vector ***L*** has been renamed as BOM:

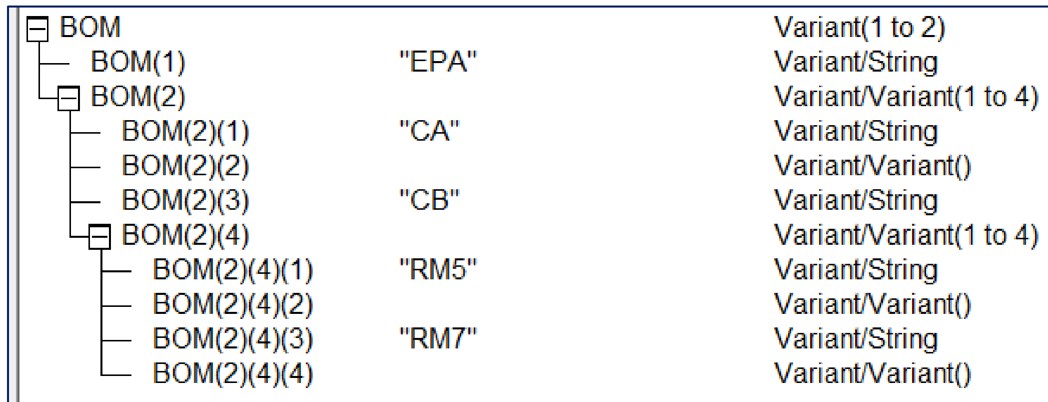

Fig. 1.10 (b). *The codification of end-product EPA using a jagged array in VBA*

The full code, with many comments is reported below:

```
Public Sub Make_BOM(ID As String)
' At first we generate an array with two elements
Dim BOM(1 To 2) As Variant
   BOM(1) = ID
   BOM(2) = Add_Branch(ID) 'Call to the recursive procedure that create the BOM
   Call Print_Bom(BOM) 'Recursive procedure
End Sub

Private Function Add_Branch(ID As String) As Variant
Dim Branch() As Variant
Dim I As Integer, N_P As Integer
Dim Rcs As Recordset2
Dim MySQL As String
   'The SQL needed to filter the BOMS table
   MySQL = "SELECT * FROM BOMS WHERE ID_Component1 = '" & CStr(ID) & "'"
   Set Rcs = CurrentDb.OpenRecordset(MySQL)
   'If some records are returned we are not at a leaf, we need to add another branch
   If Not Rcs.EOF Then
      Rcs.MoveLast
      N_P = Rcs.RecordCount
      'We need to add a branch with 2×N_P elements
      ReDim Branch(1 To 2 * N_P)
      I = 2
      Rcs.MoveFirst
      Do While Not Rcs.EOF
```

```
        'We write the code of the components in odd position
        Branch(I - 1) = Rcs.Fields(1)
        'We call recursively Add_Branch to add, if needed, a new branch in pair position
        Branch(I) = Add_Branch(CStr(Rcs.Fields(1)))
        I = I + 2
        Rcs.MoveNext
      Loop
      Set Rcs = Nothing
   End If
   Add_Branch = Branch
End Function

Private Sub Print_Bom(BOM As Variant, Optional Space As Integer = 0)
Dim B As Variant
Dim S As String
On Error Resume Next
   For Each B In BOM
      If Is_Vector(B) Then 'If single element, then print (escape condition)
        Call Print_Bom(B, Space + 4) 'Else recursive function, with increment of spaces
      Else
        S = String(Space, " ") 'Generate a string made of "Space" spaces :)
      Debug.Print S & CStr(B)
      End If
   Next B
End Sub

Private Function Is_Vector(Vector As Variant) As Boolean
'Check if Vector is a vector using an error trapping logic
On Error GoTo Err:
   Is_Vector = CBool(UBound(Vector))
Err:
   If Err > 0 Then Is_Vector = False
End Function
```

Now, invoking the procedure in the following way:

```
Call Make_BOM("EPA")
```

we get the following result:

```
CAll Make_Bom("EPA")
EPA
     CA
     CB
          RM5
          RM7
```

Fig. 1.11 *The obtained solution printed on the VBA "immediate window"*

Please note that the query MySQL used to populate the recordset Rcs with the following instruction:

```
Set Rcs = CurrentDb.OpenRecordset(MySQL)
```

operates on the following tables:

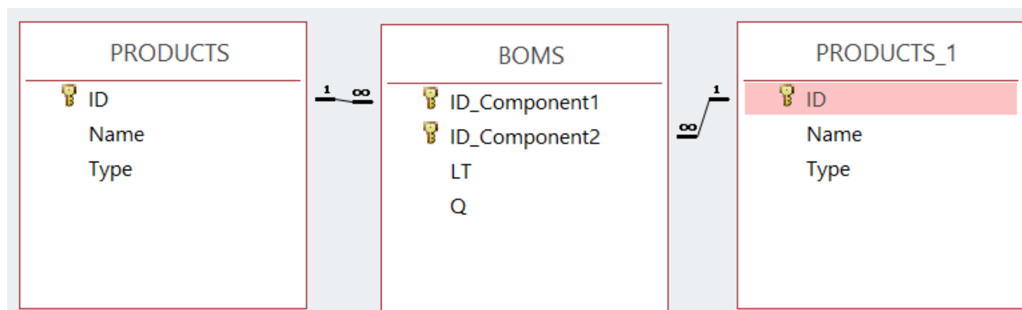

Fig. 1.12 *The tables defining the BOM hierarch*

This is because, using Access it is not possible to create, directly, two OTM relationships linking a single table (in this case PRODUCTS) and a bridge-table (in this case BOMS). In order to recreate this double-linked structure, there is the need to create a virtual copy (i.e., an alias) of the original table and, next, to connect both the original and the aliased tables with the bridge table, using two OTM relationships. In other words, we need to operate as the original and the aliased table were two distinct entities.

Also note that, in order to filter the BOMS table, so as to return only the components that form a specific level of the BOM the MySQL query has the following structure:

SELECT *

FROM BOMS

WHERE ID_Component1 = [*ID of the father component here*]

For example, to get the components forming the second level of the BOM, we should substitute the parameter [*ID of the father component here*] with 'EPA.'