Contents

Chapter 4
Conceptual Data Warehouse Design

The advantages of using conceptual models for designing databases are well known. Conceptual models facilitate communication between users and designers since they do not require knowledge about specific features of the underlying implementation platform. Further, schemas developed using conceptual models can be mapped to various logical models, such as relational, object-relational, or object-oriented models, thus simplifying responses to changes in the technology used. Moreover, conceptual models facilitate the database maintenance and evolution, since they focus on users' requirements; as a consequence, they provide better support for subsequent changes in the logical and physical schemas.

In this chapter, we focus our study on conceptual modeling for data warehouses. In particular, we base our presentation in the MultiDim model, which can be used to represent the data requirements of data warehouse and OLAP applications. The definition of the model is given in Sect. 4.1. Since hierarchies are essential for exploiting data warehouse and OLAP systems to their full capabilities, in Sect. 4.2 we consider various kinds of hierarchies that exist in real-world situations. We classify these hierarchies, giving a graphical representation of them and emphasizing the differences between them. We also present advanced aspects of conceptual modeling in Sect. 4.3. Finally, in Sect. 4.4, we revisit the OLAP operations that we presented in Chap. **??** by addressing a set of queries to the Northwind data warehouse.

4.1 Conceptual Modeling of Data Warehouses

As studied in Chap. **??**, the conventional database design process includes the creation of database schemas at the conceptual, logical, and physical levels. A **conceptual schema** is a concise description of the users' data requirements without taking into account implementation details. Conventional databases are generally designed at the conceptual level using some variation of the

well-known entity-relationship (ER) model, although the Unified Modeling Language (UML) is being increasingly used. Conceptual schemas can be easily translated to the relational model by applying a set of mapping rules.

Within the database community, it has been acknowledged that conceptual models allow better communication between designers and users for the purpose of understanding application requirements. A conceptual schema is more stable than an implementation-oriented (logical) schema, which must be changed whenever the target platform changes. Conceptual models also provide better support for visual user interfaces; for example, ER models have been very successful with users due to their intuitiveness.

However, there is no well-established and universally adopted conceptual model for multidimensional data. Due to this lack of a generic, user-friendly, and comprehensible conceptual data model, data warehouse design is usually directly performed at the logical level, based on star and/or snowflake schemas (which we will study in Chap. 5), leading to schemas that are difficult to understand by a typical user. Providing extensions to the ER and the UML models for data warehouses is not really a solution to the problem, since ultimately they represent a reflection and visualization of the underlying relational technology concepts and, in addition, reveal their own problems. Therefore, conceptual data warehousing modeling requires a model that clearly stands on top of the logical level.

In this chapter we use the MultiDim model, which is powerful enough to represent at the conceptual level all elements required in data warehouse and OLAP applications, that is, dimensions, hierarchies, and facts with associated measures. The graphical notation of the MultiDim model is given in Appendix **??**.

In order to give a general overview of the model, we shall use the example in Fig. 4.1, which illustrates the conceptual schema of the Northwind data warehouse. This figure includes several types of hierarchies, which will be presented in more detail in the subsequent sections. We next introduce the main components of the model.

A **schema** is composed of a set of dimensions and a set of facts.

A **dimension** is composed of either one level, or one or more hierarchies. A hierarchy is in turn composed of a set of levels (we explain below the notation for hierarchies). There is no graphical element to represent a dimension, it is depicted by means of its constituent elements.

A **level** is analogous to an entity type in the ER model. It describes a set of real-world concepts that, from the application perspective, have similar characteristics. Instances of a level are called **members**. For example, Product and Category are some of the levels in Fig. 4.1. As shown in the figure, a level has a set of **attributes** that describe the characteristics of their members. In addition, a level has one or several **identifiers** that uniquely identify the members of a level, each identifier being composed of one or several attributes. For example, in Fig. 4.1, CategoryID is an identifier of the Category level. Each attribute of a level has a type, that is, a domain for its values.

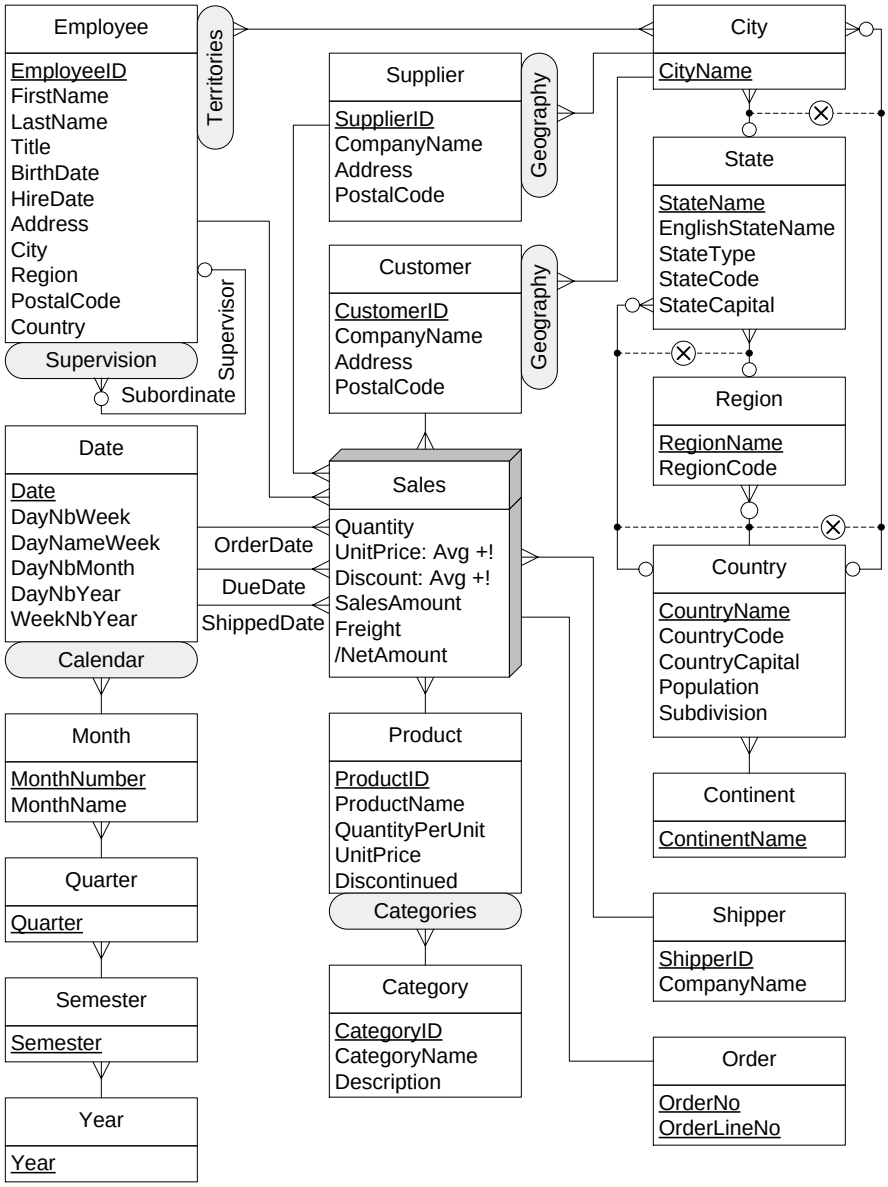


Fig. 4.1. Conceptual schema of the Northwind data warehouse

Typical value domains are integer, real, and string. We do not include type information for attributes in the graphical representation of our conceptual schemas.

A **fact** relates several levels. For example, the Sales fact in Fig. 4.1 relates the Employee, Customer, Supplier, Shipper, Order, Product, and Date levels. The same level can participate several times in a fact, playing different **roles**. Each role is identified by a name and is represented by a separate link between the level and the fact. For example, in Fig. 4.1 the Date level participates in the Sales fact with the roles OrderDate, DueDate, and ShippedDate. Instances of a fact are called **fact members**. The **cardinality** of the relationship between facts and levels, indicates the minimum and the maximum number of fact members that can be related to level members. For example, in Fig. 4.1 the Sales fact is related to the Product level with a one-to-many cardinality, which means that one sale is related to only one product and that each product can have many sales. On the other hand, the Sales fact is related to the Order level with a one-to-one cardinality, which means that a sale is related to only one order line and that an order line has only one sale.

A fact may contain attributes commonly called **measures**. These contain data (usually numerical) that are analyzed using the various perspectives represented by the dimensions. For example, the Sales fact in Fig. 4.1 includes the measures Quantity, UnitPrice, Discount, SalesAmount, Freight, and NetAmount. The identifier attributes of the levels involved in a fact indicate the granularity of the measures, that is, the level of detail at which measures are represented.

Measures are aggregated along dimensions when performing roll-up operations. As shown in Fig. 4.1, the aggregation function associated to a measure can be specified next to the measure name, where the SUM aggregation function is assumed by default. In Chap. **??** we classified measures as **additive**, **semiadditive**, or **nonadditive**. We assume by default that measures are additive, that is, they can be summarized along all dimensions. For semiadditive and nonadditive measures, we include the symbols '+!' and '$\neq$', respectively. For example, in Fig. 4.1 the measures Quantity and UnitPrice are, respectively, additive and semiadditive measures. Further, measures and level attributes may be **derived**, where they are calculated on the basis of other measures or attributes in the schema. We use the symbol '/' for indicating derived measures and attributes. For example, in Fig. 4.1 the measure NetAmount is derived.

A **hierarchy** comprises several related levels. Given two related levels of a hierarchy, the lower level is called the **child** and the higher level is called the **parent**. Thus, the relationships composing hierarchies are called **parent-child relationships**. The **cardinalities** in parent-child relationships indicate the minimum and the maximum number of members in one level that can be related to a member in another level. For example, in Fig. 4.1 the child level Product is related to the parent level Category with a one-to-many cardinality, which means that a product belongs to only one category and that a category can have many products. Notice that we do not represent the All level for the various hierarchies since this would add unnecessary complexity to conceptual multidimensional schemas.

A dimension may contain several hierarchies, each one expressing a particular criterion used for analysis purposes; thus, we include the **hierarchy name** to differentiate them. For example, in Fig. 4.1, the Employee dimensions has two hierarchies, namely, Territories and Supervision. If a single level contains attributes forming a hierarchy, such as the attributes City, Region, and Country in the Employee dimension in Fig. 4.1, this means that the user is not interested in employing this hierarchy for aggregation purposes.

Levels in a hierarchy are used to analyze data at various *granularities*, or levels of detail. For example, the Product level contains specific information about products, while the Category level may be used to see these products from the more general perspective of the categories to which they belong. The level in a hierarchy that contains the most detailed data is called the **leaf level**. The name of the leaf level defines the dimension name, except for the case where the same level participates several times in a fact, in which case the role name defines the dimension name. These are called **role-playing dimensions**. The level in a hierarchy representing the most general data is called the **root level**. It is usual (but not mandatory) to represent the root of a hierarchy using a distinguished level called All, which contains a single member, denoted all. The decision of including this level in multidimensional schemas is left to the designer. In the remainder, we do not show the All level in the hierarchies (except when we consider it necessary for clarity of presentation), since we consider that it is meaningless in conceptual schemas.

The identifier attributes of a parent level define how child members are grouped. For example, in Fig. 4.1, CategoryID in the Category level is an identifier attribute; it is used for grouping different product members during the roll-up operation from the Product to the Category levels. However, in the case of many-to-many parent-child relationships, it is also needed to determine how to distribute the measures from a child to its parent members. For this, a **distributing attribute** may be used. For example, in Fig. 4.1, the relationship between Employee and City is many-to-many, that is, the same employee can be assigned to several cities. A distributing attribute can be used to store the percentage of time that an employee devotes to each city. Another example of a distributing attribute is given in Fig. 4.13.

Finally, it is sometimes the case that two or more parent-child relationships are **exclusive**. This is represented using the symbol '⊗'. An example is given in Fig. 4.1, where states can be aggregated either into regions or into countries. Thus, according to their type, states participate in only one of the relationships departing from the State level.

The reader may have noticed that many of the concepts of the MultiDim model are similar to those used in Chap. **??**, when we presented the multidimensional model and the data cube. This suggests that the MultiDim model stays on top of the logical level, hiding from the user the implementation details. In other words, the model represents a conceptual data cube. Therefore, we will call the schema in Fig. 4.1 as the Northwind data cube.

4.2 Hierarchies

Hierarchies are key elements in analytical applications, since they provide the means to represent the data under analysis at different abstraction levels. In real-world situations, users must deal with complex hierarchies of various kinds. Even though we can model complex hierarchies at a conceptual level, as we will study in this section, logical models of data warehouse and OLAP systems only provide a limited set of kinds of hierarchies. Therefore, users are often unable to capture the essential semantics of multidimensional applications and must limit their analysis to considering only the predefined kinds of hierarchies provided by the tools in use. Nevertheless, a data warehouse designer should be aware of the problems that the various kinds of hierarchies introduce and be able to deal with them. In this section, we discuss several kinds of hierarchies that can be represented by means of the MultiDim model, although the classification of hierarchies that we will provide is independent of the conceptual model used to represent them. Since many of the hierarchies we study next are not present in the Northwind data cube of Fig. 4.1, we will introduce new ad hoc examples when needed.

4.2.1 Balanced Hierarchies

A **balanced hierarchy** has only one path at the schema level, where all levels are mandatory. An example is given by hierarchy Product → Category in Fig. 4.1. At the instance level, the members form a tree where all the branches have the same length, as shown in Fig. **??**. All parent members have at least one child member, and a child member belongs exactly to one parent member. For example, in Fig. **??** each category is assigned at least one product, and a product belongs to only one category.

4.2.2 Unbalanced Hierarchies

An **unbalanced hierarchy** has only one path at the schema level, where at least one level is not mandatory. Therefore, at the instance level there can be parent members without associated child members. Figure 4.2a shows a hierarchy schema in which a bank is composed of several branches, where a branch may have agencies; further an agency may have ATMs. As a consequence, at the instance level the members represent an unbalanced tree, that is, the branches of the tree have different lengths, since some parent members do not have associated child members. For example, Fig. 4.2b shows a branch with no agency, and several agencies with no ATM. As in the case of balanced hierarchies, the cardinalities in the schema imply that every child member

should belong to at most one parent member. For example, in Fig. 4.2 every agency belongs to one branch.

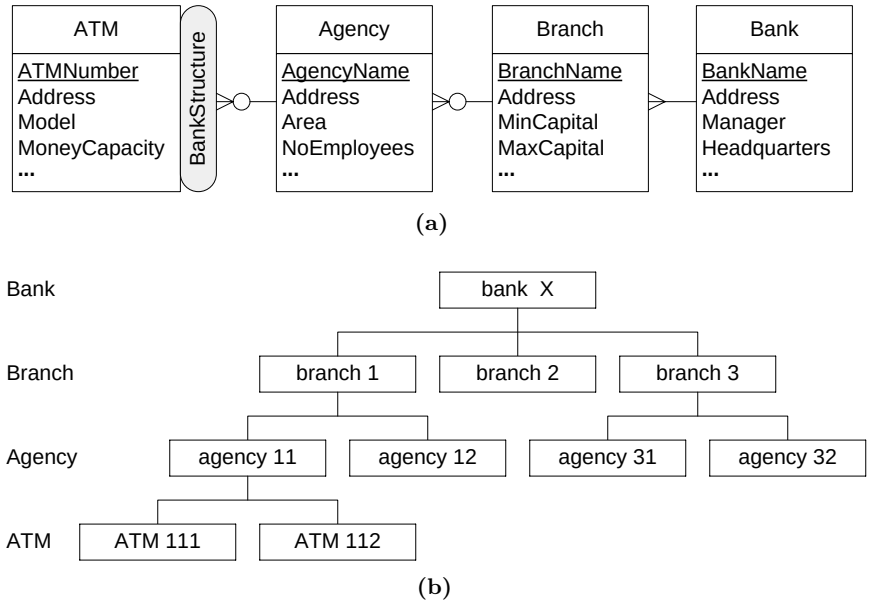


Fig. 4.2. An unbalanced hierarchy. (**a**) Schema; (**b**) Examples of instances

Unbalanced hierarchies include a special case that we call **recursive hierarchies**, also called **parent-child hierarchies**. In this kind of hierarchy, the same level is linked by the two roles of a parent-child relationship (note the difference between the notions of parent-child hierarchies and relationships). An example is given by dimension Employee in Fig. 4.1, which represents an organizational chart in terms of the employee-supervisor relationship. The Subordinate and Supervisor roles of the parent-child relationship are linked to the Employee level. As seen in Fig. 4.3, this hierarchy is unbalanced since employees with no subordinate will not have descendants in the instance tree.

4.2.3 Generalized Hierarchies

Sometimes, the members of a level are of different types. A typical example arises with customers, which can be either companies or persons. Such a situation is usually captured in an ER model using the generalization relationship studied in Chap. **??**. Further, suppose that measures pertaining to customers must be aggregated differently according to the customer type,

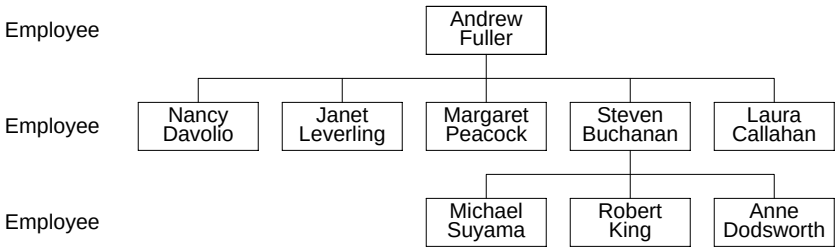


Fig. 4.3. Instances of the parent-child hierarchy in the Northwind data warehouse

where for companies the aggregation path is Customer → Sector → Branch, while for persons it is Customer → Profession → Branch. To represent such kinds of hierarchies, the MultiDim model has the graphical notation shown in Fig. 4.4a, where the common and specific hierarchy levels and also the parent-child relationships between them are clearly represented. Such hierarchies are called **generalized hierarchies**.

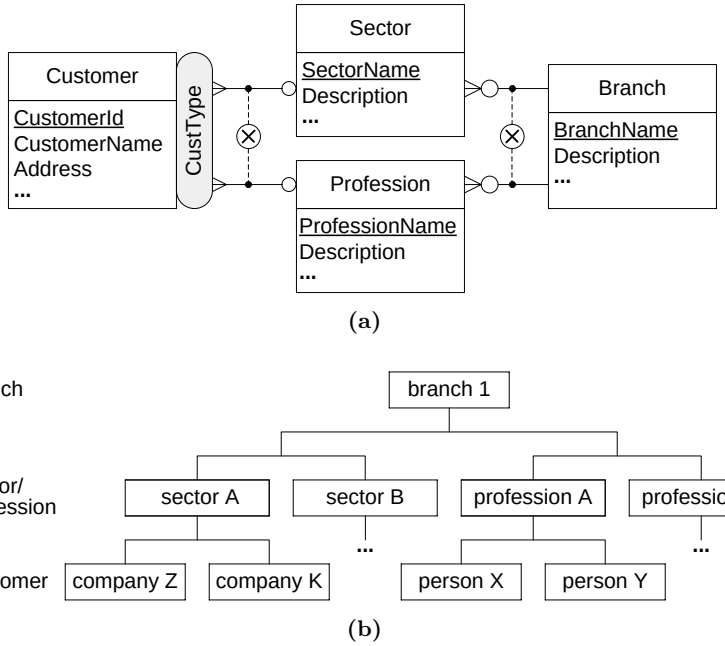


Fig. 4.4. A generalized hierarchy. (**a**) Schema; (**b**) Examples of instances

At the schema level, a generalized hierarchy contains multiple exclusive paths sharing at least the leaf level; they may also share some other levels,

as shown in Fig. 4.4a. This figure shows the two aggregation paths described above, one for each type of customer, where both belong to the same hierarchy. At the instance level, each member of the hierarchy belongs to only one path, as can be seen in Fig. 4.4b. We use the symbol '⊗' to indicate that the paths are exclusive for every member. Such a notation is equivalent to the *xor* annotation used in UML. The levels at which the alternative paths split and join are called, respectively, the **splitting** and **joining levels**.

The distinction between splitting and joining levels is important to ensure correct measure aggregation during roll-up operations, a property called summarizability, which we discussed in Chap. **??**. Generalized hierarchies are, in general, not summarizable. For example, not all customers are mapped to the Profession level. Thus, the aggregation mechanism should be modified when a splitting level is reached in a roll-up operation.

In generalized hierarchies, it is not necessary that splitting levels are joined. An example is the hierarchy in Fig. 4.5, which is used for analyzing international publications. Three kinds of publications are considered: journals, books, and conference proceedings. The latter can be aggregated to the conference level. However, there is not a common joining level for all paths.

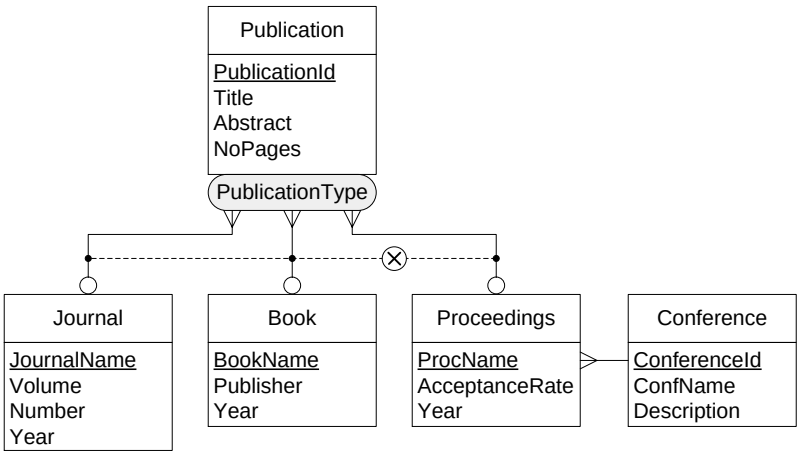


Fig. 4.5. A generalized hierarchy without a joining level

Generalized hierarchies include a special case commonly referred to as **ragged** hierarchies. An example is the hierarchy City → State → Region → Country → Continent given in Fig. 4.1. As can be seen in Fig. 4.6, some countries, such as Belgium, are divided into regions, whereas others, such as Germany, are not. Furthermore, small countries like the Vatican have neither regions nor states. A ragged hierarchy is a generalized hierarchy where alternative paths are obtained by skipping one or several intermediate levels. At the instance level, every child member has only one parent member, although

the path length from the leaves to the same parent level can be different for different members.

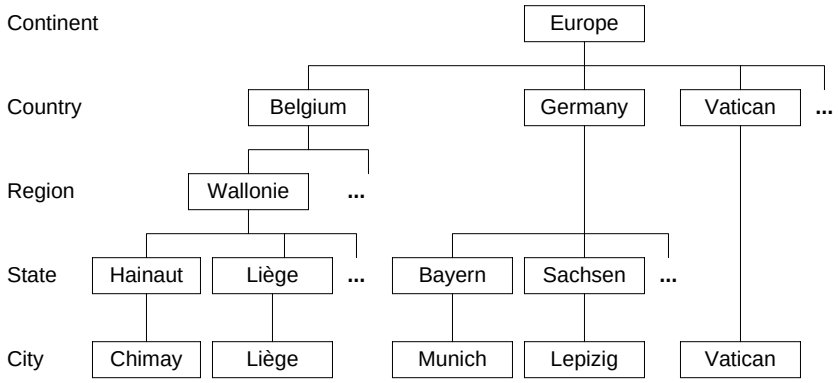


Fig. 4.6. Examples of instances of the ragged hierarchy in Fig. 4.1

4.2.4 Alternative Hierarchies

Alternative hierarchies represent the situation where at the schema level there are several nonexclusive hierarchies that share at least the leaf level. An example is given in Fig. 4.7a, where the Date dimension includes two hierarchies corresponding to different groupings of months into calendar years and fiscal years. Figure 4.7b shows an instance of the dimension (we do not show members of the Date level), where it is supposed that fiscal years begin in February. As it can be seen, the hierarchies form a graph, since a child member is associated with more than one parent member, and these parent members belong to different levels. Alternative hierarchies are needed when we want to analyze measures from a unique perspective (e.g., time) using alternative aggregations.

Note the difference between generalized and alternative hierarchies (see Figs. 4.4 and 4.7). Although the two kinds of hierarchies share some levels, they represent different situations. In a generalized hierarchy, a child member is related to *only one* of the paths, whereas in an alternative hierarchy a child member is related to *all paths*, and the user must choose one of them for analysis.

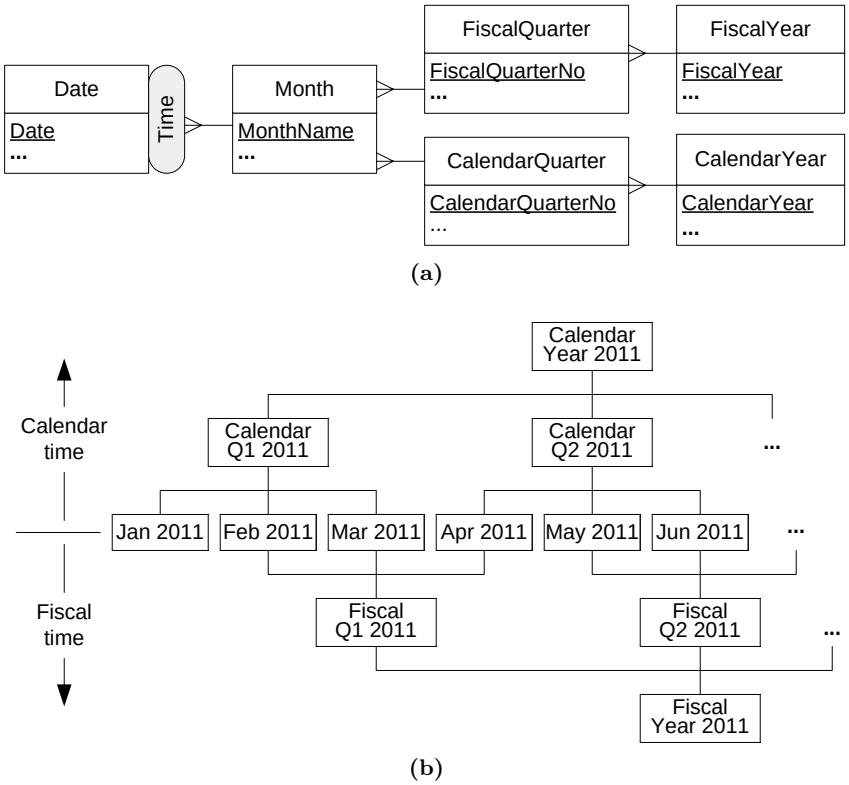


Fig. 4.7. An alternative hierarchy. (**a**) Schema; (**b**) Examples of instances

4.2.5 Parallel Hierarchies

Parallel hierarchies arise when a dimension has several hierarchies associated with it, accounting for different analysis criteria. Further, the component hierarchies may be of different kinds.

Parallel hierarchies can be **dependent** or **independent** depending on whether the component hierarchies share levels. Figure 4.8 shows an example of a dimension that has two parallel independent hierarchies. The hierarchy ProductGroups is used for grouping products according to categories or departments, while the hierarchy DistributorLocation groups them according to distributors' divisions or regions. On the other hand, the parallel dependent hierarchies given in Fig. 4.9 represent a company that requires sales analysis for stores located in several countries. The hierarchy StoreLocation represents the geographic division of the store address, while the hierarchy SalesOrganization represents the organizational division of the company. Since the two hierarchies share the State level, this level plays different roles according to

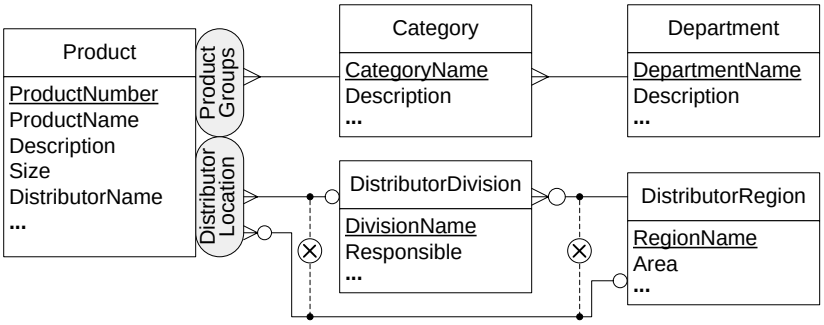


Fig. 4.8. An example of parallel independent hierarchies

the hierarchy chosen for the analysis. Sharing levels in a conceptual schema reduces the number of its elements without losing its semantics, thus improving readability. In order to unambiguously define the levels composing the various hierarchies, the hierarchy name must be included in the sharing level for hierarchies that *continue beyond* that level. This is the case of StoreLocation and SalesOrganization indicated on level State.

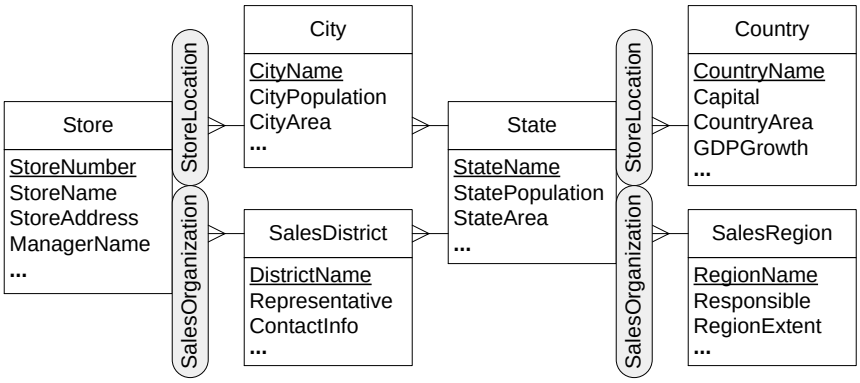


Fig. 4.9. An example of parallel dependent hierarchies

Even though both alternative and parallel hierarchies share some levels and may be composed of several hierarchies, they represent different situations and should be clearly distinguishable at the conceptual level. This is done by including only one (for alternative hierarchies) or several (for parallel hierarchies) hierarchy names, which account for various analysis criteria. In this way, the user is aware that in the case of alternative hierarchies it is not meaningful to combine levels from different component hierarchies, whereas this can be done for parallel hierarchies. For example, for the schema in

Fig. 4.9 the user can safely issue a query "Sales figures for stores in city A that belong to the sales district B."

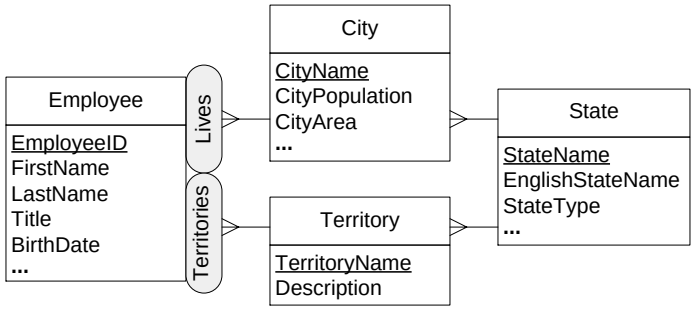


Fig. 4.10. Parallel dependent hierarchies leading to different parent members of the shared level

Further, in parallel dependent hierarchies a leaf member may be related to various different members in a shared level, which is not the case for alternative hierarchies that share levels. For instance, consider the schema in Fig. 4.10, which refers to the living place and the territory assignment of sales employees. It should be obvious that traversing the hierarchies Lives and Territory from the Employee to the State level will lead to different states for employees who live in one state and are assigned to another. As a consequence of this, aggregated measure values can be reused for shared levels in alternative hierarchies, whereas this is not the case for parallel dependent hierarchies. For example, suppose that the amount of sales generated by employees E1, E2, and E3 are \$50, \$100, and \$150, respectively. If all employees live in state A, but only E1 and E2 work in this state, aggregating the sales of all employees to the State level following the Lives hierarchy gives a total amount of \$300, whereas the corresponding value will be equal to \$150 when the Territories hierarchy is traversed. Note that both results are correct, since the two hierarchies represent different analysis criteria.

4.2.6 Nonstrict Hierarchies

In the hierarchies studied so far, we have assumed that each parent-child relationship has a one-to-many cardinality, that is, a child member is related to at most one parent member and a parent member may be related to several child members. However, many-to-many relationships between parent and child levels are very common in real-life applications. For example, a diagnosis may belong to several diagnosis groups, 1 week may span 2 months, a product may be classified into various categories, etc.

A hierarchy that has *at least* one many-to-many relationship is called **nonstrict**, otherwise it is called **strict**. The fact that a hierarchy is strict or not is orthogonal to its kind. Thus, the hierarchies previously presented can be either strict or nonstrict. We next analyze some issues that arise when dealing with nonstrict hierarchies.

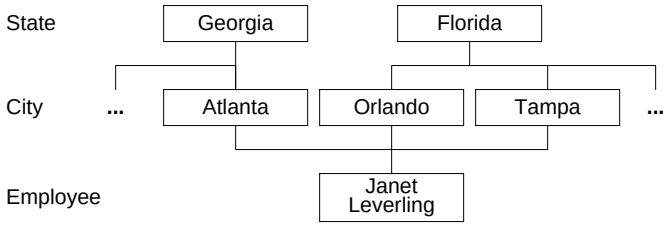


Fig. 4.11. Examples of instances of the nonstrict hierarchy in Fig. 4.1

Figure 4.1 shows a nonstrict hierarchy where an employee may be assigned to several cities. Some instances of this hierarchy are shown in Fig. 4.11, where the employee Janet Leverling is assigned to three cities that belong to two states. Since at the instance level a child member may have more than one parent member, the members of the hierarchy form an acyclic graph. Note the slight abuse of terminology. We use the term "nonstrict hierarchy" to denote an acyclic classification graph for several reasons. The term "hierarchy" conveys the notion that users need to analyze measures at different levels of detail; the term "acyclic classification graph" is less clear in this sense. Further, the term "hierarchy" is used by practitioners and is customary in data warehouse research.

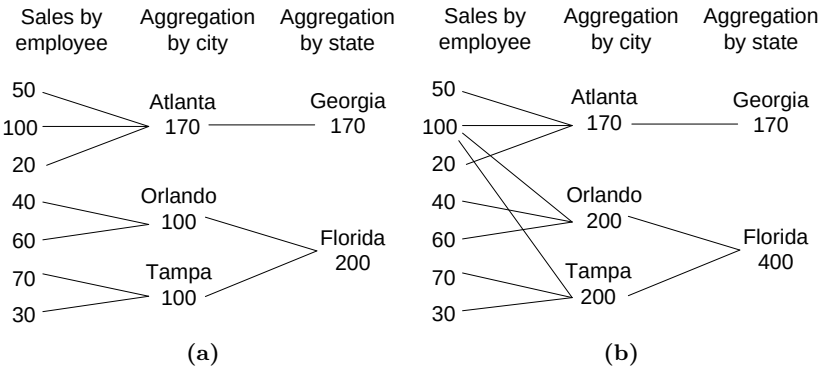


Fig. 4.12. Double-counting problem when aggregating a sales amount measure in Fig. 4.11. (**a**) Strict hierarchy; (**b**) Nonstrict hierarchy

Nonstrict hierarchies induce the problem of **double counting** of measures when a roll-up operation reaches a many-to-many relationship. Let us consider the example in Fig. 4.12, which illustrates sales by employees with aggregations along City and State levels (defined in Fig. 4.11), and employee Janet Leverling with total sales equal to 100. Figure 4.12a shows a situation where the employee has been assigned to Atlanta, in a strict hierarchy scenario. The sum of sales by territory and by state can be calculated straightforwardly, as the figure shows. Figure 4.12b shows a nonstrict hierarchy scenario, where the employee has been assigned the territories Atlanta, Orlando, and Tampa. This approach causes incorrect aggregated results, since the employee's sales are counted three times instead of only once.

One solution to the double-counting problem consists in transforming a nonstrict hierarchy into a strict one by creating a new member for each set of parent members participating in a many-to-many relationship. In our example, a new member that represents the three cities Atlanta, Orlando, and Tampa will be created. However, in this case a new member must also be created in the state level, since the three cities belong to two states. Another solution would be to ignore the existence of several parent members and to choose one of them as the primary member. For example, we may choose the city of Atlanta. However, neither of these solutions correspond to the users' analysis requirements, since in the former, artificial categories are introduced, and in the latter, some pertinent analysis scenarios are ignored.

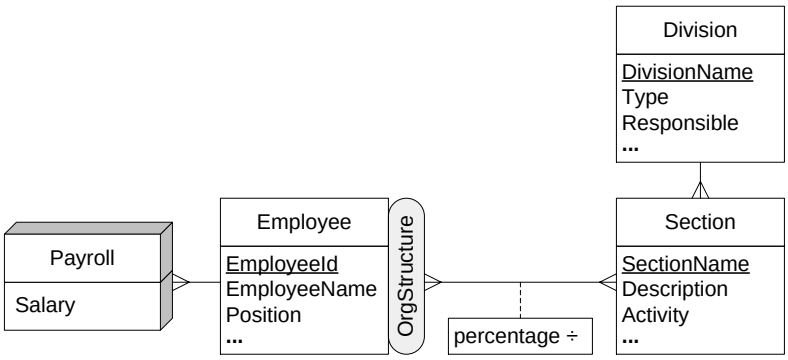


Fig. 4.13. A nonstrict hierarchy with a distributing attribute

An alternative approach to the double-counting problem would be to indicate how measures are distributed between several parent members for many-to-many relationships. For example, Fig. 4.13 shows a nonstrict hierarchy where employees may work in several sections. The schema includes a measure that represents an employee's overall salary, that is, the sum of the salaries paid in each section. Suppose that an attribute stores the percentage of time for which an employee works in each section. In this case, we depict

this attribute in the relationship with an additional symbol '÷' indicating that it is a **distributing attribute** determining how measures are divided between several parent members in a many-to-many relationship.

Choosing an appropriate distributing attribute is important in order to avoid approximate results when aggregating measures. For example, suppose that in Fig. 4.13 the distributing attribute represents the percentage of time that an employee works in a specific section. If the employee has a higher position in one section, although she works less time in that section she may earn a higher salary. Thus, applying the percentage of time as a distributing attribute for measures representing an employee's overall salary may not give an exact result. Note also that in cases where the distributing attribute is unknown, it can be approximated by considering the total number of parent members with which the child member is associated. In the example of Fig. 4.12, since Janet Leverling is associated with three cities, one third of the value of the measure will be accounted for each city.

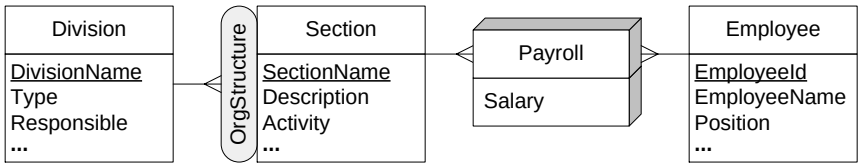


Fig. 4.14. Transforming a nonstrict hierarchy into a strict one with an additional dimension

Figure 4.14 shows another solution to the problem of Fig. 4.13 where we transformed a nonstrict hierarchy into independent dimensions. However, this solution corresponds to a different conceptual schema, where the focus of analysis has been changed from employees' salaries to employees' salaries by section. Note that this solution can only be applied when the exact distribution of the measures is known, for instance, when the amounts of salary paid for working in the different sections are known. It cannot be applied to nonstrict hierarchies without a distributing attribute, as in Fig. 4.11.

Nevertheless, although the solution in Fig. 4.14 aggregates correctly the Salary measure when applying the roll-up operation from the Section to the Division levels, the problem of double counting of the same employee will occur. Suppose that we want to use the schema in Fig. 4.14 to calculate the number of employees by section or by division; this value can be calculated by counting the instances of employees in the fact. The example in Fig. 4.15a considers five employees who are assigned to various sections. Counting the number of employees who work in each section gives correct results. However, the aggregated values for each section cannot be reused for calculating the number of employees in every division, since some employees (E1 and E2 in

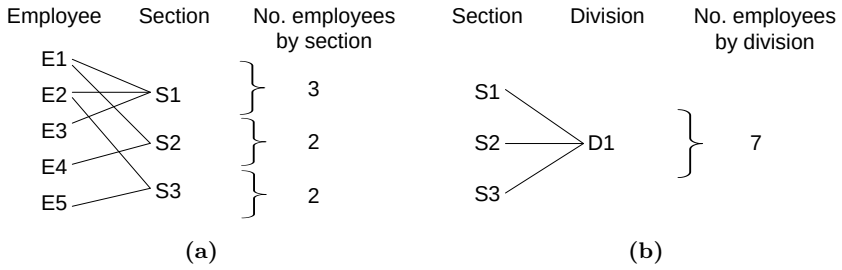


Fig. 4.15. Double-counting problem for a nonstrict hierarchy

Fig. 4.15a) will be counted twice and the total result will give a value equal to 7 (Fig. 4.15b) instead of 5.

In summary, nonstrict hierarchies can be handled in several ways:

- Transforming a nonstrict hierarchy into a strict one:
  - Creating a new parent member for each group of parent members linked to a single child member in a many-to-many relationship.
  - Choosing one parent member as the primary member and ignoring the existence of other parent members.
  - Replacing the nonstrict hierarchy by two independent dimensions.
- Including a distributing attribute.
- Calculating approximate values of a distributing attribute.

Since each solution has its advantages and disadvantages and requires special aggregation procedures, the designer must select the appropriate solution according to the situation at hand and users' requirements.

4.3 Advanced Modeling Aspects

In this section, we discuss particular modeling issues, namely, facts with multiple granularities, many-to-many dimensions, and links between facts, and show how they can be represented in the MultiDim model.

4.3.1 Facts with Multiple Granularities

Sometimes it is the case that measures are captured at **multiple granularities**. An example is given in Fig. 4.16, where, for instance, sales for USA might be reported per state, while European sales might be reported per

city. As another example, consider a medical data warehouse for analyzing patients, where there is a diagnosis dimension with levels diagnosis, diagnosis family, and diagnosis group. A patient may be related to a diagnosis at the lowest granularity, but may also have (more imprecise) diagnoses at the diagnosis family and diagnosis group levels.

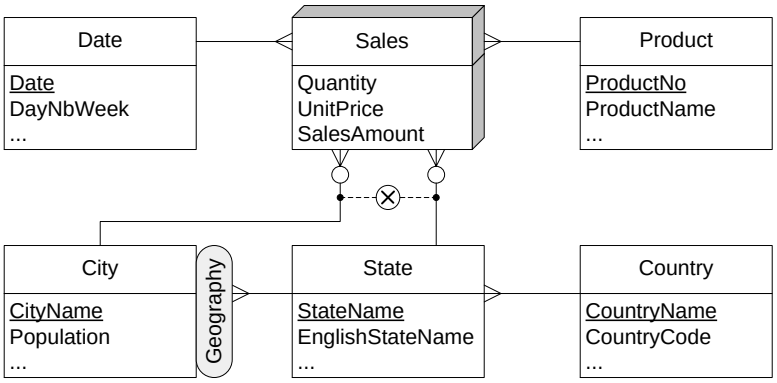


Fig. 4.16. Multiple granularities for a Sales fact

As can be seen in Fig. 4.16, this situation can be modeled using exclusive relationships between the various granularity levels. Obviously, the issue in this case is to get correct analysis results when fact data are registered at multiple granularities.

4.3.2 Many-to-Many Dimensions

In a **many-to-many dimension**, several members of the dimension participate in the same fact member. An example is shown in Fig. 4.17. Since an account can be jointly owned by several clients, aggregation of the balance according to the clients will count this balance as many times as the number of account holders. For example, as shown in Fig. 4.18, suppose that at some point at date D1 there are two accounts A1 and A2 with balances of, respectively, 100 and 500. Suppose further that both accounts are shared between several clients: account A1 is shared by C1, C2, and C3, and account A2 by C1 and C2. The total balance of the two accounts is equal to 600; however, aggregation (e.g., according to the Date or the Client dimension) gives a value equal to 1,300.

The problem of **double counting** introduced above can be analyzed through the concept of **multidimensional normal forms** (MNFs). MNFs determine the conditions that ensure correct measure aggregation in the pres-

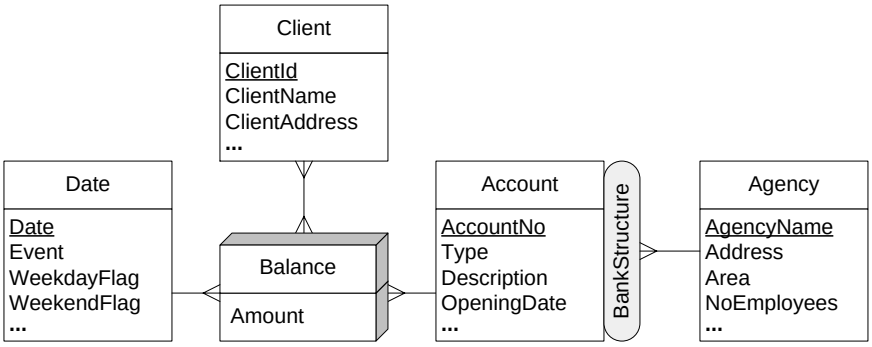


Fig. 4.17. Multidimensional schema for the analysis of bank accounts

ence of the complex hierarchies studied in this chapter. The first multidimensional normal form (1MNF) requires each measure to be uniquely identified by the set of associated leaf levels. The 1MNF is the basis for correct schema design. To analyze the schema in Fig. 4.17 in terms of the 1MNF, we need to find out the functional dependencies that exist between the leaf levels and the measures. Since the balance depends on the specific account and the time when it is considered, the account and the time determine the balance. Therefore, the schema in Fig. 4.17 does not satisfy the 1MNF, since the measure is not determined by all leaf levels, and thus the fact must be decomposed.

Date	Account	Client	Balance
D1	A1	C1	100
D1	A1	C2	100
D1	A1	C3	100
D1	A2	C1	500
D1	A2	C2	500

Fig. 4.18. An example of double-counting problem in a many-to-many dimension

Let us recall the notion of multivalued dependency we have seen in Chap. **??**. There are two possible ways in which the Balance fact in Fig. 4.17 can be decomposed. In the first one, the same joint account may have different clients assigned to it during different periods of time, and thus the time and the account multidetermine the clients. This situation leads to the solution shown in Fig. 4.19a, where the original fact is decomposed into two facts, that is, AccountHolders and Balance. If the joint account holders do not change over time, clients are multidetermined just by the accounts (but not the date). In this case, the link relating the Date level and the AccountHolders fact can be eliminated. Alternatively, this situation can be modeled with a nonstrict hierarchy as shown in Fig. 4.19b.

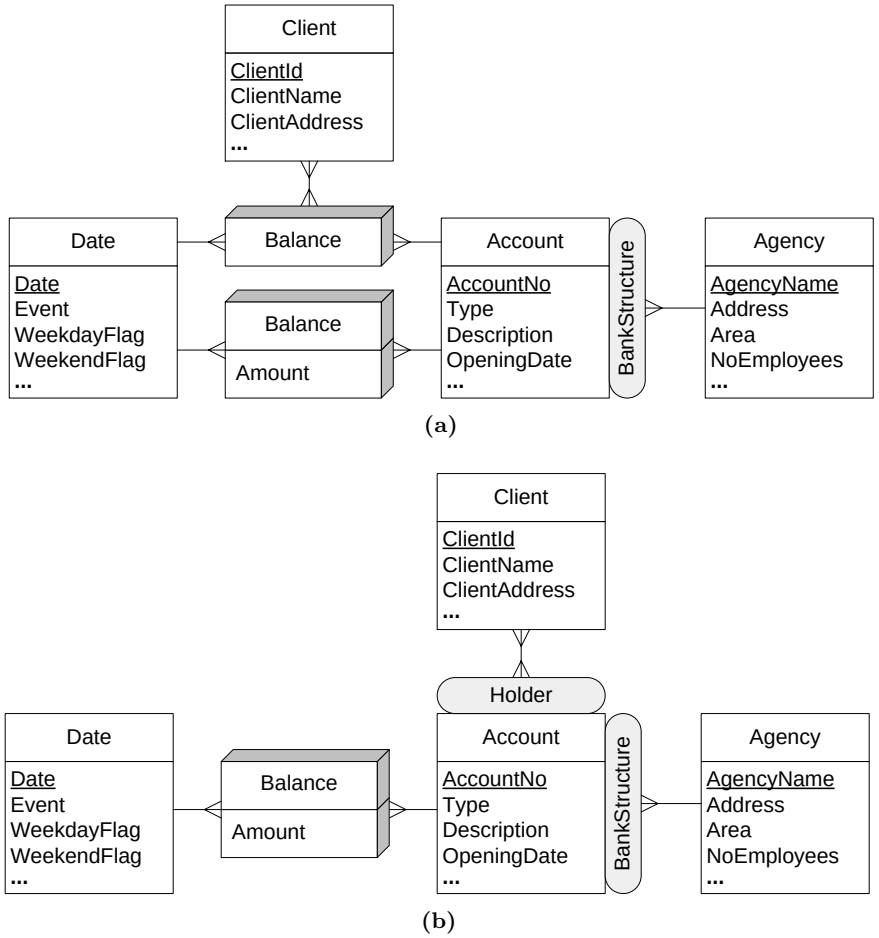


Fig. 4.19. Two possible decompositions of the fact in Fig. 4.17. (**a**) Creating two facts; (**b**) Including a nonstrict hierarchy

Even though the solutions proposed in Fig. 4.19 eliminate the double-counting problem, the two schemas in Fig. 4.19 require programming effort for queries that ask for information about individual clients. The difference lies in the fact that in Fig. 4.19a a drill-across operation (see Sect. **??**) between the two facts is needed, while in Fig. 4.19b special procedures for aggregation in nonstrict hierarchies must be applied. In the case of Fig. 4.19a, since the two facts represent different granularities, queries with drill-across operations are complex, demanding a conversion either from a finer to a coarser granularity (e.g., grouping clients to know who holds a specific balance in an account) or vice versa (e.g., distributing a balance between different account holders). Note also that the two schemas in Fig. 4.19 could represent

the information about the percentage of ownership of accounts by customers (if this is known). This could be represented by a measure in the AccountHolders fact in Fig. 4.19a and by a distributing attribute in the many-to-many relationship in Fig. 4.19b.

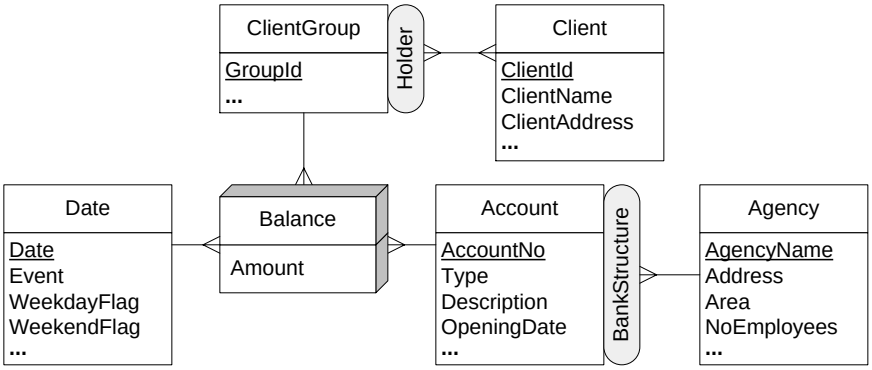


Fig. 4.20. Alternative decomposition of the fact in Fig. 4.17

Another solution to this problem is shown in Fig. 4.20. In this solution, an additional level is created, which represents the groups of clients participating in joint accounts. In the case of the example in Fig. 4.18, two groups should be created: one that includes clients C1, C2, and C3, and another with clients C1 and C2. Note, however, that the schema in Fig. 4.20 is not in the 1MNF, since the measure Balance is not determined by all leaf levels, that is, it is only determined by Date and Account. Therefore, the schema must be decomposed leading to schemas similar to those in Fig. 4.19, with the difference that in this case the Client level in the two schemas in Fig. 4.19 is replaced by a nonstrict hierarchy composed of the ClientGroup and the Client levels.

Finally, to avoid many-to-many dimensions we can choose one client as the primary account owner and ignore the other clients. In this way, only one client will be related to a specific balance, and the schema in Fig. 4.17 can be used without any problems related to double counting of measures. However, this solution may not represent the real-world situation and may exclude from the analysis the other clients of joint accounts.

In summary, many-to-many dimensions in multidimensional schemas can be avoided by using one of the solutions presented in Fig. 4.19. The choice between these alternatives depends on the functional and multivalued dependencies existing in the fact, the kinds of hierarchies in the schema, and the complexity of the implementation.

4.3.3 Links between Facts

Sometimes it is necessary to define a link between two related facts even if they share dimensions. Fig. 4.21 shows an example where the facts Order and Delivery share dimensions Customer and Date, while each fact has specific dimensions, that is, Employee in Order and Shipper in Delivery. As indicated by the link between the two facts, suppose that several orders can be delivered by a single delivery and that a single order (e.g., containing many products) can be delivered by several deliveries.

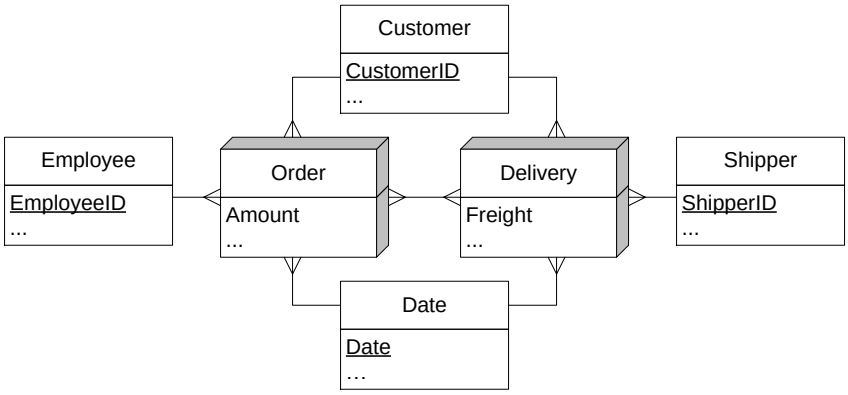


Fig. 4.21. An excerpt of a conceptual schema for analyzing orders and deliveries

Order	Date	Customer	Employee	Amount
O1	1/5/2014	C1	E1	250.00
O2	1/5/2014	C1	E1	147.00
O3	3/5/2014	C1	E2	133.00
O4	5/5/2014	C2	E2	190.00

Order	Delivery
O1	D1
O3	D1
O2	D2
O2	D3
O4	D4

Delivery	Date	Customer	Shipper	Freight
D1	6/5/2014	C1	S3	25.00
D2	7/5/2014	C1	S2	30.00
D3	8/5/2014	C1	S2	27.00
D4	9/5/2014	C2	S1	50.00

Fig. 4.22. Examples of instances of the facts and their link in Fig. 4.21

Possible instances of the above facts and their link are shown in Fig. 4.22. Notice that, even if the facts share dimensions, an explicit link between the

facts is needed to keep the information of how orders were delivered. Indeed, neither Customer nor Date can be used for this purpose.

The cardinalities of the links between facts can be one-to-one, one-to-many, and many-to-many. In the example above, we consider the latter case. If instead the cardinality were one-to-many, in this case an order is delivered by only one delivery, but one delivery may concern multiple orders.

The usual way to combine cubes is through the drill-across operation (see Sect. **??**), which performs the join through their common dimensions. However, as we have seen, such operation is not appropriate in this example. Therefore, a new version of the drill-across operation that takes into account the link between the facts is needed. The result of such drill-across is given in Fig. 4.23.

Order	OrderDate	Customer	Employee	Amount	Delivery	DeliveryDate	Shipper	Freight
O1	1/5/2014	C1	E1	250.00	D1	6/5/2014	S3	25.00
O3	3/5/2014	C1	E2	133.00	D1	6/5/2014	S3	25.00
O2	1/5/2014	C1	E1	147.00	D2	7/5/2014	S2	30.00
O2	1/5/2014	C1	E1	147.00	D3	8/5/2014	S2	27.00
O4	5/5/2014	C2	E2	190.00	D4	9/5/2014	S1	50.00

Fig. 4.23. Drill-across of the facts through their link in Fig. 4.22

4.4 Querying the Northwind Cube Using the OLAP Operations

We conclude the chapter showing how the OLAP operations studied in Chap. **??** can be used to express queries over a conceptual schema, independently of the actual underlying implementation. We use for this the Northwind cube in Fig. 4.1.

Query 4.1. *Total sales amount per customer, year, and product category.*

ROLLUP*(Sales, Customer → Customer, OrderDate → Year,
Product → Category, SUM(SalesAmount))

The ROLLUP* operation is used to specify the levels at which each of the dimensions Customer, OrderDate, and Product are rolled-up. For the other dimensions in the cube a roll-up to All is performed. The SUM operation is applied to aggregate the measure SalesAmount. All other measures of the cube are removed from the result.

Query 4.2. *Yearly sales amount for each pair of customer country and supplier countries.*

```
ROLLUP*(Sales, OrderDate → Year, Customer → Country,
        Supplier → Country, SUM(SalesAmount))
```

As in the previous query, a roll-up to the specified levels is performed, while performing a SUM operation to aggregate the measure SalesAmount.

Query 4.3. *Monthly sales by customer state compared to those of the previous year.*

```
Sales1 ← ROLLUP*(Sales, OrderDate → Month, Customer → State,
         SUM(SalesAmount))
Sales2 ← RENAME(Sales1, SalesAmount → PrevYearSalesAmount)
Result ← DRILLACROSS(Sales2, Sales1,
         Sales2.OrderDate.Month = Sales1.OrderDate.Month AND
         Sales2.OrderDate.Year+1 = Sales1.OrderDate.Year AND
         Sales2.Customer.State = Sales1.Customer.State)
```

Here, we first apply a ROLLUP operation to aggregate the measure SalesAmount. Then, a copy of the resulting cube, with the measure renamed as PrevYearSalesAmount, is kept in the cube Sales2. The two cubes are joined with the DRILLACROSS operation, where the join condition ensures that cells corresponding to the same month of two consecutive years and to the same client state are merged in a single cell in the result. Although we include the join condition for the Customer dimension, since it is an equijoin, this is not mandatory, it is assumed by default for all the dimensions not mentioned in the join condition. In the following, we do not include the equijoins in the conditions in the DRILLACROSS operations.

Query 4.4. *Monthly sales growth per product, that is, total sales per product compared to those of the previous month.*

```
Sales1 ← ROLLUP*(Sales, OrderDate → Month, Product → Product,
         SUM(SalesAmount))
Sales2 ← RENAME(Sales1, SalesAmount → PrevMonthSalesAmount)
Sales3 ← DRILLACROSS(Sales2, Sales1,
         ( Sales1.OrderDate.Month > 1 AND
           Sales2.OrderDate.Month+1 = Sales1.OrderDate.Month AND
           Sales2.OrderDate.Year = Sales1.OrderDate.Year ) OR
         ( Sales1.OrderDate.Month = 1 AND Sales2.OrderDate.Month = 12 AND
           Sales2.OrderDate.Year+1 = Sales1.OrderDate.Year ) )
Result ← ADDMEASURE(Sales3, SalesGrowth =
         SalesAmount - PrevMonthSalesAmount )
```

As in the previous query, we first apply a ROLLUP operation, make a copy of the resulting cube, and join the two cubes with the DRILLACROSS operation. However, here the join condition is more involved than in the previous query, since two cases must be considered. In the first one, for the months starting from February (Month > 1) the cells to be merged must be consecutive and belong to the same year. In the second case, the cell corresponding to January must be merged with the one of December from the previous year. Finally, in the last step we compute a new measure SalesGrowth as the difference between the sales amount of the two corresponding months.

Query 4.5. *Three best-selling employees.*

```
Sales1 ← ROLLUP*(Sales, Employee → Employee, SUM(SalesAmount))
Result ← MAX(Sales1, SalesAmount, 3)
```

Here, we roll-up all the dimensions of the cube, except Employee, to the All level, while aggregating the measure SalesAmount. Then, the MAX operation is applied while specifying that cells with the top three values of the measure are kept in the result.

Query 4.6. *Best-selling employee per product and year.*

```
Sales1 ← ROLLUP*(Sales, Employee → Employee,
        Product → Product, OrderDate → Year, SUM(SalesAmount))
Result ← MAX(Sales1, SalesAmount) BY Product, OrderDate
```

In this query, we roll-up the dimensions of the cube as specified. Then, the MAX operation is applied after grouping by Product and OrderDate.

Query 4.7. *Countries that account for top 50% of the sales amount.*

```
Sales1 ← ROLLUP*(Sales, Customer → Country, SUM(SalesAmount))
Result ← TOPPERCENT(Sales1, Customer, 50) ORDER BY SalesAmount DESC
```

Here, we roll-up the Customer dimension to Country level and the other dimensions to the All level. Then, the TOPPERCENT operation selects the countries that cumulatively account for top 50% of the sales amount.

Query 4.8. *Total sales and average monthly sales by employee and year.*

```
Sales1 ← ROLLUP*(Sales, Employee → Employee, OrderDate → Month,
        SUM(SalesAmount))
Result ← ROLLUP*(Sales1, Employee → Employee, OrderDate → Year,
        SUM(SalesAmount), AVG(SalesAmount))
```

Here, we first roll-up the cube to the Employee and Month levels by summing the SalesAmount measure. Then, we perform a second roll-up to the Year level to obtain to overall sales and the average of monthly sales.

Query 4.9. *Total sales amount and total discount amount per product and month.*

```
Sales1 ← ADDMEASURE(Sales, TotalDisc = Discount * Quantity * UnitPrice)
Result ← ROLLUP*(Sales1, Product → Product, OrderDate → Month,
        SUM(SalesAmount), SUM(TotalDisc))
```

Here, we first compute a new measure TotalDisc from three other measures. Then, we roll-up the cube to the Product and Month levels.

Query 4.10. *Monthly year-to-date sales for each product category.*

```
Sales1 ← ROLLUP*(Sales, Product → Category, OrderDate → Month,
         SUM(SalesAmount))
Result ← ADDMEASURE(Sales1, YTD = SUM(SalesAmount) OVER
         OrderDate BY Year ALL CELLS PRECEDING)
```

Here, we start by performing a roll-up to the category and month levels. Then, a new measure is created by applying the SUM aggregation function to a window composed of all preceding cells of the same year. Notice that it is supposed that the members of the Date dimension are ordered according to the calendar time.

Query 4.11. *Moving average over the last 3 months of the sales amount by product category.*

```
Sales1 ← ROLLUP*(Sales, Product → Category, OrderDate → Month,
         SUM(SalesAmount))
Result ← ADDMEASURE(Sales1, MovAvg = AVG(SalesAmount) OVER
         OrderDate 2 CELLS PRECEDING)
```

In the first roll-up, we aggregate the SalesAmount measure by category and month. Then, we compute the moving average over a window containing the cells corresponding to the current month and the two preceding months.

Query 4.12. *Personal sales amount made by an employee compared with the total sales amount made by herself and her subordinates during 1997.*

```
Sales1 ← SLICE(Sales, OrderDate.Year = 1997)
Sales2 ← ROLLUP*(Sales1, Employee → Employee, SUM(SalesAmount))
Sales3 ← RENAME(Sales2, SalesAmount → PersonalSales)
Sales4 ← RECROLLUP(Sales2, Employee → Employee, Supervision,
         SUM(SalesAmount))
Result ← DRILLACROSS(Sales4, Sales3)
```

In the first step, we restrict the data in the cube to the year 1997. Then, in the second step we perform the aggregation of the sales amount measure by employee, thus obtaining the sales figures independently of the supervision hierarchy. Then, in the third step the obtained measure is renamed. In the fourth step, we apply the recursive roll-up, which performs an iteration over the supervision hierarchy by aggregating children to parent until the top level is reached. Finally, the last step obtains the cube with both measures.

Query 4.13. *Total sales amount, number of products, and sum of the quantities sold for each order.*

```
ROLLUP*(Sales, Order → Order, SUM(SalesAmount),
         COUNT(Product) AS ProductCount, SUM(Quantity))
```

Here, we roll-up all the dimensions, except Order, to the All level, while adding the SalesAmount and Quantity measures and counting the number of products.

Query 4.14. *For each month, total number of orders, total sales amount, and average sales amount by order.*

```
Sales1 ← ROLLUP*(Sales, OrderDate → Month, Order → Order,
         SUM(SalesAmount))
Result ← ROLLUP*(Sales1, OrderDate → Month, SUM(SalesAmount),
         AVG(SalesAmount) AS AvgSales, COUNT(Order) AS OrderCount)
```

In the query above, we first roll-up to the Month and Order levels. Then, we perform another roll-up to remove the Order dimension and obtain the requested measures.

Query 4.15. *For each employee, total sales amount, number of cities, and number of states to which she is assigned.*

```
ROLLUP*(Sales, Employee → State, SUM(SalesAmount), COUNT(DISTINCT City)
         AS NoCities, COUNT(DISTINCT State) AS NoStates)
```

Recall that Territories is a nonstrict hierarchy in the Employee dimension. In this query, we roll-up to the State level while adding the SalesAmount measure and counting the number of distinct cities and states. Notice that the ROLLUP* operation takes into account the fact that the hierarchy is nonstrict and avoids the double-counting problem to which we referred in Sect. 4.2.6.

4.5 Summary

This chapter focused on conceptual modeling for data warehouses. As it the case for databases, conceptual modeling allows user requirements to be represented while hiding actual implementation details, that is, regardless of the actual underlying data representation. To explain conceptual multidimensional modeling we used the MultiDim model, which is based on the entity-relationship model and provides an intuitive graphical notation. It is well known that graphical representations facilitate the understanding of application requirements by users and designers.

We have presented a comprehensive classification of hierarchies, taking into account their differences at the schema and at the instance level. We started by describing balanced, unbalanced, and generalized hierarchies, all of which account for a single analysis criterion. Recursive (or parent-child) and ragged hierarchies are special cases of unbalanced and generalized hierarchies, respectively. Then, we introduced alternative hierarchies, which are composed of several hierarchies defining various aggregation paths for the same analysis criterion. We continued with parallel hierarchies, which are composed of several hierarchies accounting for different analysis criteria. When parallel hierarchies share a level, they are called dependent, otherwise they are called independent. All the above hierarchies can be either strict or nonstrict, depending on whether they contain many-to-many relationships between parent and child levels. Nonstrict hierarchies define graphs at the

instance level. We then presented advanced modeling aspects, namely, facts with multiple granularities and many-to-many dimensions. These often arise in practice but are frequently overlooked in the data warehouse literature. In Chap. 5, we will study how all these concepts can be implemented at the logical level. We concluded showing how the OLAP operations introduced in Chap. **??** can be applied over the conceptual model, using as example a set of queries over the Northwind data cube.

4.6 Bibliographic Notes

Conceptual data warehouse design was first introduced by Golfarelli et al. [6]. A detailed description of conceptual multidimensional models can be found in [33]. Many multidimensional models have been proposed in the literature. Some of them provide graphical representations based on the ER model (e.g., [32, 35]), as is the case of the MultiDim model, while others are based on UML (e.g., [1, 20, 34]). Other models propose new notations (e.g., [7, 14, 36]), while others do not refer to a graphical representation (e.g., [12, 25, 27]). There is great variation in the kinds of hierarchies supported by current multidimensional models. A detailed comparison of how the various multidimensional models cope with hierarchies is given in [21, 23]. The inclusion of explicit links between cubes in multidimensional models was proposed in [31]. Multidimensional normal forms were defined in [17, 18].

The Object Management Group (OMG) has proposed the Common Warehouse Model (CWM)[1] as a standard for representing data warehouse and OLAP systems. This model provides a framework for representing metadata about data sources, data targets, transformations, and analysis, in addition to processes and operations for the creation and management of warehouse data. The CWM model is represented as a layered structure consisting of a number of submodels. One of these submodels, the resource layer, defines models that can be used for representing data in data warehouses and includes the relational model as one of them. Further, the analysis layer presents a metamodel for OLAP, which includes the concepts of a dimension and a hierarchy. In the CWM, it is possible to represent all of the kinds of hierarchies presented in this chapter.

[1] http://www.omg.org/docs/formal/03-03-02.pdf

Chapter 5
Logical Data Warehouse Design

Conceptual models are useful to design database applications since they favor the communication between the stakeholders in a project. However, conceptual models must be translated into logical ones for their implementation on a database management system. In this chapter, we study how the conceptual multidimensional model studied in the previous chapter can be represented in the relational model. We start in Sect. 5.1 by describing the three logical models for data warehouses, namely, relational OLAP (ROLAP), multidimensional OLAP (MOLAP), and hybrid OLAP (HOLAP). In Sect. 5.2, we focus on the relational representation of data warehouses and study four typical implementations: the star, snowflake, starflake, and constellation schemas. In Sect. 5.3, we present the rules for mapping a conceptual multidimensional model (in our case, the MultiDim model) to the relational model. Section 5.4 discusses how to represent the time dimension. Sections 5.5 and 5.6 study how hierarchies, facts with multiple granularities, and many-to-many dimensions can be implemented in the relational model. Section 5.7 is devoted to the study of slowly changing dimensions, which arise when dimensions in a data warehouse are updated. In Sect. 5.8, we study how a data cube can be represented in the relational model and how it can be queried in SQL using the SQL/OLAP extension. Finally, to show how these concepts are applied in practice, in Sects. 5.9 and 5.10 we show how the Northwind cube can be implemented, respectively, in Microsoft Analysis Services and in Mondrian.

5.1 Logical Modeling of Data Warehouses

There are several approaches for implementing a multidimensional model, depending on how the data cube is stored. These are described next.

Relational OLAP (**ROLAP**) systems store multidimensional data in relational databases and support extensions to SQL and special access methods to efficiently implement the OLAP operations. Further, in order to increase performance, aggregates are precomputed in relational tables (we will study aggregate computation in Chap. **??**). These aggregates, together with indexing structures, take a large space from the database. The advantages of ROLAP systems rely on the fact that relational databases are well standardized and provide a large storage capacity. However, since OLAP operations must be performed on relational tables, this usually yields complex SQL queries.

Multidimensional OLAP (**MOLAP**) systems store data in specialized multidimensional data structures (e.g., arrays), which are combined with hashing and indexing techniques. Therefore, the OLAP operations can be implemented efficiently, since such operations are very natural and simple to perform. MOLAP systems generally provides less storage capacity than ROLAP systems. Furthermore, MOLAP systems are proprietary, which reduces their portability.

Hybrid OLAP (**HOLAP**) systems combine the two previous approaches in order to benefit from the storage capacity of ROLAP and the processing capabilities of MOLAP. For example, a HOLAP server may store large volumes of detailed data in a relational database, while aggregations are kept in a separate MOLAP store.

Current OLAP tools support a combination of the above models. Nevertheless, most of these tools rely on an underlying data warehouse implemented on a relational database management system. For this reason, in what follows, we study the ROLAP implementation in detail.

5.2 Relational Data Warehouse Design

One possible relational representation of the multidimensional model is based on the **star schema**, where there is one central **fact table**, and a set of **dimension tables**, one for each dimension. An example is given in Fig. 5.1, where the fact table is depicted in gray and the dimension tables are depicted in white. The fact table contains the foreign keys of the related dimension tables, namely, ProductKey, StoreKey, PromotionKey, and DateKey, and the measures, namely, Amount and Quantity. As shown in the figure, **referential integrity** constraints are specified between the fact table and each of the dimension tables.

In a star schema, the dimension tables are, in general, not normalized. Therefore, there may contain redundant data, especially in the presence of hierarchies. This is the case for dimension Product in Fig. 5.1 since all products belonging to the same category will have redundant information for the attributes describing the category and the department. The same occurs in dimension Store with the attributes describing the city and the state.

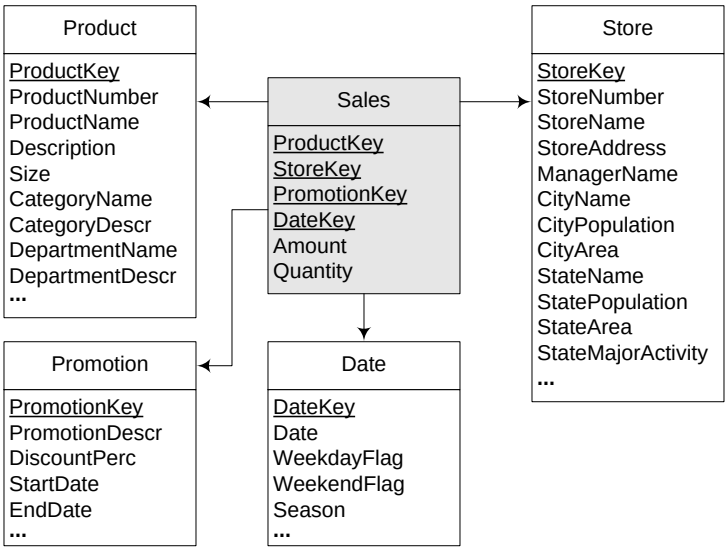


Fig. 5.1. An example of a star schema

On the other hand, fact tables are usually normalized: their key is the union of the foreign keys since this union functionally determines all the measures, while there is no functional dependency between the foreign key attributes. In Fig. 5.1, the fact table Sales is normalized and its key is composed by ProductKey, StoreKey, PromotionKey, and DateKey.

A **snowflake schema** avoids the redundancy of star schemas by normalizing the dimension tables. Therefore, a dimension is represented by several tables related by **referential integrity** constraints. In addition, as in the case of star schemas, referential integrity constraints also relate the fact table and the dimension tables at the finest level of detail.

An example of a snowflake schema is given in Fig. 5.2. Here, the fact table is exactly the same as in Fig. 5.1. However, the dimensions Product and Store are now represented by normalized tables. For example, in the Product dimension, the information about categories has been moved to the table Category, and only the attribute CategoryKey remained in the original table. Thus, only the value of this key is repeated for each product of the same category, but the information about a category will only be stored once, in table Category. Normalized tables are easy to maintain and optimize storage space. However, performance is affected since more joins need to be performed when executing queries that require hierarchies to be traversed. For example, the query "Total sales by category" for the star schema in Fig. 5.1 reads in SQL as follows:

```
SELECT    CategoryName, SUM(Amount)
FROM      Product P, Sales S
```

Fig. 5.2. An example of a snowflake schema

```
WHERE     P.ProductKey = S.ProductKey
GROUP BY CategoryName
```

while in the snowflake schema in Fig. 5.2 we need an extra join, as follows:

```
SELECT    CategoryName, SUM(Amount)
FROM      Product P, Category C, Sales S
WHERE     P.ProductKey = S.ProductKey AND P.CategoryKey = C.CategoryKey
GROUP BY CategoryName
```

A **starflake schema** is a combination of the star and the snowflake schemas, where some dimensions are normalized while others are not. We would have a starflake schema if we replace the tables Product, Category, and Department in Fig. 5.2, by the dimension table Product of Fig. 5.1, and leave all other tables in Fig. 5.2 (like dimension table Store) unchanged.

Finally, a **constellation schema** has multiple fact tables that share dimension tables. The example given in Fig. 5.3 has two fact tables Sales and Purchases sharing the Date and Product dimension. Constellation schemas may include both normalized and unnormalized dimension tables.

We will discuss further star and snowflake schemas when we study logical representation of hierarchies later in this chapter.

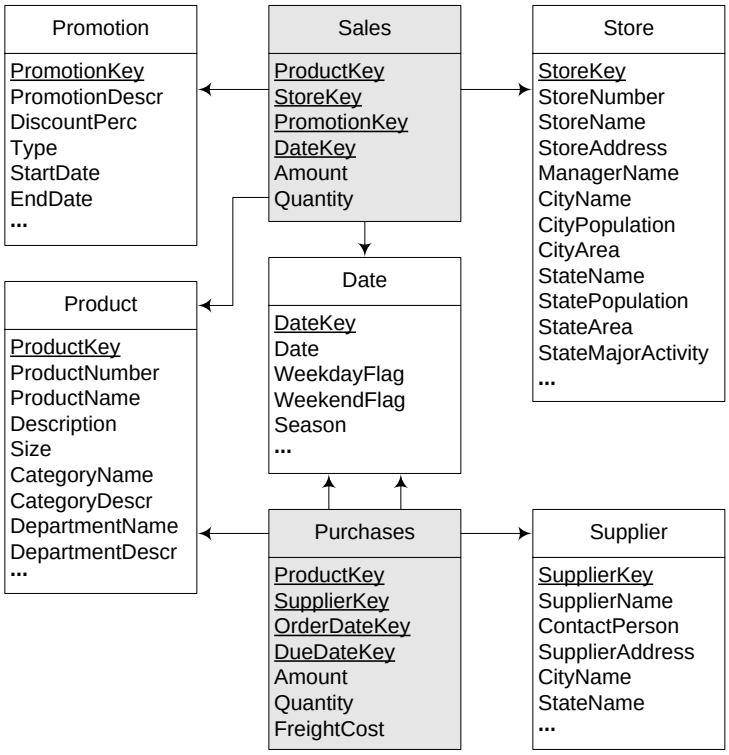


Fig. 5.3. An example of a constellation schema

5.3 Relational Representation of Data Warehouses

We define next a set of rules that are used to translate a multidimensional conceptual schema into a relational one. These rules are based on those given in Sect. **??**, which translate an ER schema into the relational model.

Rule 1: A level L, provided it is not related to a fact with a one-to-one relationship, is mapped to a table T_L that contains all attributes of the level. A surrogate key may be added to the table, otherwise the identifier of the level will be the key of the table. Note that additional attributes will be added to this table when mapping relationships using Rule 3 below.

Rule 2: A fact F is mapped to a table T_F that includes as attributes all measures of the fact. Further, a surrogate key may be added to the table. Note that additional attributes will be added to this table when mapping relationships using Rule 3 below.

Rule 3: A relationship between either a fact F and a dimension level L, or between dimension levels L_P and L_C (standing for the parent and child

levels, respectively), can be mapped in three different ways, depending on its cardinalities:

Rule 3a: If the relationship is one-to-one, the table corresponding to the fact (T_F) or to the child level (T_C) is extended with all the attributes of the dimension level or the parent level, respectively.

Rule 3b: If the relationship is one-to-many, the table corresponding to the fact (T_F) or to the child level (T_C) is extended with the surrogate key of the table corresponding to the dimension level (T_L) or the parent level (T_P), respectively, that is, there is a foreign key in the fact or child table pointing to the other table.

Rule 3c: If the relationship is many-to-many, a new table T_B (standing for bridge table) is created that contains as attributes the surrogate keys of the tables corresponding to the fact (T_F) and the dimension level (T_L), or the parent (T_P) and child levels (T_C), respectively. If the relationship has a distributing attribute, an additional attribute is added to the table to store this information.

In the above rules, surrogate keys are generated for each dimension level in a data warehouse. The main reason for this is to provide independence from the keys of the underlying source systems because such keys can change across time. Another advantage of this solution is that surrogate keys are usually represented as integers in order to increase efficiency, whereas keys from source systems may be represented in less efficient data types such as strings. Nevertheless, the keys coming from the source systems should also be kept in the dimensions to be able to match data from sources with data in the warehouse. Obviously, an alternative solution is to reuse the keys from the source systems in the data warehouse.

Notice that a fact table obtained by the mapping rules above will contain the surrogate key of each level related to the fact with a one-to-many relationship, one for each role that the level is playing. The key of the table is composed of the surrogate keys of all the participating levels. Alternatively, if a surrogate key is added to the fact table, the combination of the surrogate keys of all the participating levels becomes an alternate key.

As we will see in Sect. 5.5, more specialized rules are needed for mapping the various kinds of hierarchies that we studied in Chap. 4.

Applying the above rules to the Northwind conceptual data cube given in Fig. 4.1 yields the tables shown in Fig. 5.4. The Sales table includes eight foreign keys, that is, one for each level related to the fact with a one-to-many relationship. Recall from Chap. 4 that in **role-playing dimensions**, a dimension plays several roles. This is the case for the dimension Date where, in the relational model, each role will be represented by a foreign key. Thus, OrderDateKey, DueDateKey, and ShippedDateKey are foreign keys to the Date dimension table in Fig. 5.4. Note also that dimension Order is related to the fact with a one-to-one relationship. Therefore, the attributes of the dimension are included as part of the fact table. For this reason, such a dimension

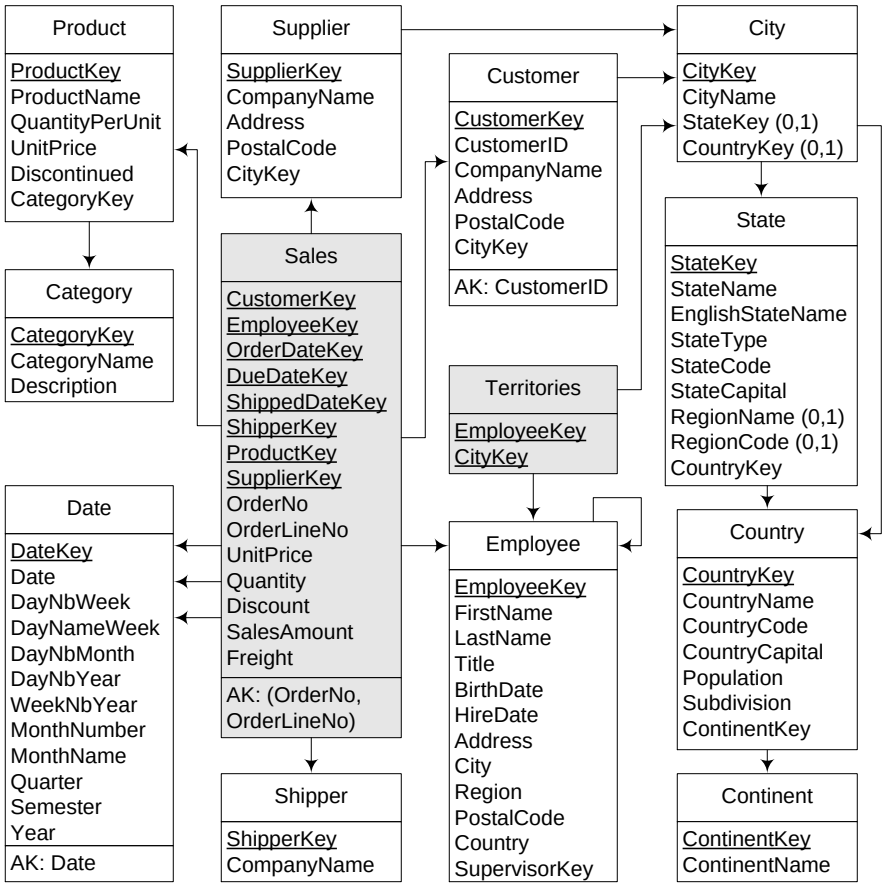


Fig. 5.4. Relational representation of the Northwind data warehouse in Fig. 4.1

is called a **fact** (or **degenerate**) **dimension**. The fact table also contains five attributes representing the measures UnitPrice, Quantity, Discount, SalesAmount, and Freight. Finally, note that the many-to-many parent-child relationship between Employee and Territory is mapped to the table Territories, containing two foreign keys.

With respect to keys, in the Northwind data warehouse of Fig. 5.4 we have illustrated the two possibilities for defining the keys of dimension levels, namely, generating surrogate keys and keeping the database key as data warehouse key. For example, Customer has a surrogate key CustomerKey and a database key CustomerID. On the other hand, SupplierKey in Supplier is a database key. The choice of one among these two solutions is addressed in the ETL process that we will see in Chap. **??**.

5.4 Time Dimension

Time information is present in almost every data warehouse. It is included both as foreign keys in fact tables, indicating the time when a fact took place, and as a time dimension, containing the aggregation levels, that is, the different ways in which facts can be aggregated across time.

In OLTP database applications, temporal information is usually derived from attributes of type DATE using the functions provided by the DBMS. For example, a typical OLTP application would not explicitly store whether a particular day is in a weekend or is a holiday: this would be computed on the fly using appropriate functions. On the other hand, such information is stored in a data warehouse as attributes in the time dimension since OLAP queries are highly demanding, and there is no time to perform such computations each time a fact must be summarized. For example, a query like "Total sales during weekends," posed over the schema of Fig. 5.1, would be easily evaluated with the following SQL query:

```
SELECT SUM(SalesAmount)
FROM   Date D, Sales S
WHERE D.DateKey = S.DateKey AND D.WeekendFlag = 1
```

The granularity of the time dimension varies depending on their use. For example, if we are interested in monthly data, we would define the time dimension with a granularity that will correspond to a month. Thus, the time dimension table of a data warehouse spanning 5 years will have $5 \times 12 = 60$ tuples. On the other hand, if we are interested in more detailed data, we could define the time dimension with a granularity that corresponds to a second. In this case, in a data warehouse spanning 5 years we will have a time dimension with $5 \times 12 \times 30 \times 24 \times 3{,}600 = 155{,}520{,}000$ tuples. The time dimension has the particularity that it can be (and in practice it is) populated automatically.

Finally, note that a time dimension may have more than one hierarchy (recall our calendar/fiscal year example in Fig. 4.7). Further, even if we use a single hierarchy we must be careful to satisfy the summarizability conditions. For example, a day aggregates correctly over a month and a year levels (a day belongs to exactly 1 month and 1 year), whereas a week may correspond to 2 different months, and thus the week level cannot be aggregated over the month level in a time dimension hierarchy.

5.5 Logical Representation of Hierarchies

The general mapping rules given in the previous section do not capture the specific semantics of all of the kinds of hierarchies described in Sect. 4.2. In addition, for some kinds of hierarchies, alternative logical representations

exist. In this section, we consider in detail the logical representation of the various kinds of hierarchies studied in Chap. 4.

5.5.1 Balanced Hierarchies

As we have seen, in a conceptual multidimensional schema, the levels of dimension hierarchies are represented independently, and these levels are linked by parent-child relationships. Therefore, applying the mapping rules given in Sect. 5.3 to balanced hierarchies leads to **snowflake schemas** described before in this chapter: each level is represented as a separate table, which includes the key and the attributes of the level, as well as foreign keys for the parent-child relationships. For example, applying Rules 1 and 3b to the Categories hierarchy in Fig. 4.1 yields a snowflake structure with tables Product and Category shown in Fig. 5.5a.

Nevertheless, if **star schemas** are required, it is necessary to represent hierarchies using flat tables, where the key and the attributes of all levels forming a hierarchy are included in a single table. This structure can be obtained by denormalizing the tables that represent several hierarchy levels. As an example, the Date dimension of Fig. 4.1 can be represented in a single table containing all attributes, as shown in Fig. 5.5b.

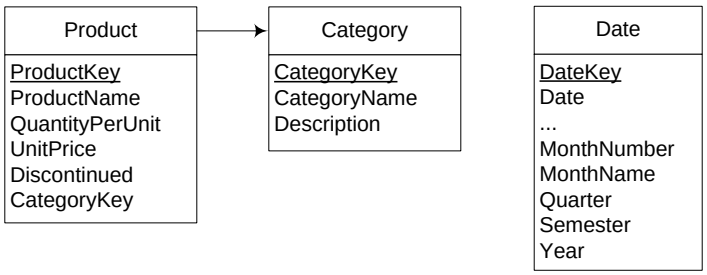


Fig. 5.5. Relations for a balanced hierarchy. (**a**) Snowflake structure; (**b**) Flat table

As we have seen in Sect. 5.2, **snowflake schemas** better represent hierarchical structures than star schemas, since every level can be easily distinguished and, further, levels can be reused between different hierarchies. Additionally, in this representation specific attributes can be included in the different levels of a hierarchy. For example, the Product and Category tables in Fig. 5.5a have specific attributes. However, snowflake schemas are less performant for querying due to the joins that are needed for combining the data scattered in the various tables composing a hierarchy.

On the other hand, **star schemas** facilitate query formulation since fewer joins are needed for expressing queries, owing to denormalization. Additionally, much research has been done to improve system performance for processing star queries. However, star schemas have some drawbacks. For example, they do not model hierarchies adequately since the hierarchy structure is not clear. For example, for the Store dimension in Fig. 5.1, it is not clear which attributes can be used for hierarchies. As can also be seen in the figure, it is difficult to clearly associate attributes with their corresponding levels, making the hierarchy structure difficult to understand.

5.5.2 Unbalanced Hierarchies

Since unbalanced hierarchies do not satisfy the summarizability conditions (see Sect. **??**), the mapping described in Sect. 5.3 may lead to the problem of excluding from the analysis the members of nonleaf levels that do not have an associated child. For instance, since in Fig. 4.2a all measures are associated with the ATM level, these measures will be aggregated into the higher levels only for those agencies that have ATMs and, similarly, only for those branches that have agencies. To avoid this problem, an unbalanced hierarchy can be transformed into a balanced one using placeholders (marked PH1, PH2, . . . , PHn in Fig. 5.6) or null values in missing levels. Then, the logical mapping for balanced hierarchies may be applied.

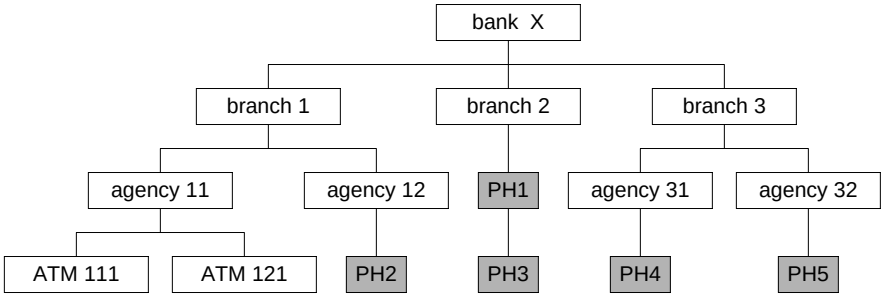


Fig. 5.6. Transformation of the unbalanced hierarchy in Fig. 4.2b into a balanced one using placeholders

The above transformation has the following consequences. First, the fact table contains measures belonging to all levels whose members can be a leaf at the instance level. For example, measures for the ATM level and for the Agency level will be included in the fact table for those members that do not have an ATM. This has the problem that users must be aware that they have to deal with fact data at several different granularities. Further, when

for a child member there are two or more consecutive parent levels missing, measure values must be repeated for aggregation purposes. For example, this would be the case for branch 2 in Fig. 5.6 since two placeholders are used for two consecutive missing levels. In addition, the introduction of meaningless values requires additional storage space. Finally, special interface must be developed to hide placeholders from users.

Recall from Sect. 4.2.2 that **parent-child hierarchies** are a special case of unbalanced hierarchies. Mapping these hierarchies to the relational model yields tables containing all attributes of a level, and an additional foreign key relating child members to their corresponding parent. For example, the table Employee in Fig. 5.4 shows the relational representation of the parent-child hierarchy in Fig. 4.1. Although such a table represents the semantics of parent-child hierarchies, operations over it are more complex. In particular, recursive queries are necessary for traversing a parent-child hierarchy. Recursive queries are allowed both in SQL and in MDX.

5.5.3 Generalized Hierarchies

Generalized hierarchies account for the case where dimension members are of different kinds, and each kind has a specific aggregation path. For example, in Fig. 4.4, customers can be either companies or persons, where companies are aggregated through the path Customer → Sector → Branch, while persons are aggregated through the path Customer → Profession → Branch.

As was the case for balanced hierarchies, two approaches can be used for representing generalized hierarchies at the logical level: create a table for each level, leading to snowflake schemas, or create a single flat table for all the levels, where null values are used for attributes that do not pertain to specific members (e.g., tuples for companies will have null values in attributes corresponding to persons). Alternatively, a mix of these two approaches can be followed: create one table for the common levels and another table for the specific ones. Finally, we could also use separate fact and dimension tables for each path. In all these approaches we must keep metadata about which tables compose the different aggregation paths, while we need to specify additional constraints to ensure correct queries (e.g., to avoid grouping Sector with Profession in Fig. 4.4).

Applying the mapping described in Sect. 5.3 to the generalized hierarchy in Fig. 4.4 yields the relations shown in Fig. 5.7. Even though this schema clearly represents the hierarchical structure, it does not allow one to traverse only the common levels of the hierarchy (e.g., to go from Customer to Branch). To ensure this possibility, we must add the following mapping rule.

Rule 4: A table corresponding to a splitting level in a generalized hierarchy has an additional attribute which is a foreign key of the next joining level,

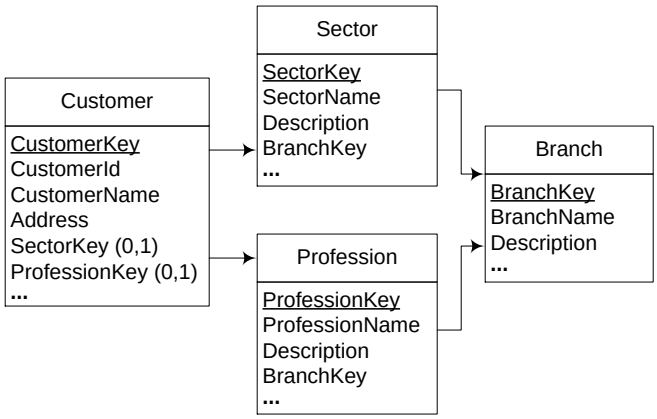


Fig. 5.7. Relations for the generalized hierarchy in Fig. 4.4

provided it exists. The table may also include a discriminating attribute that indicates the specific aggregation path of each member.

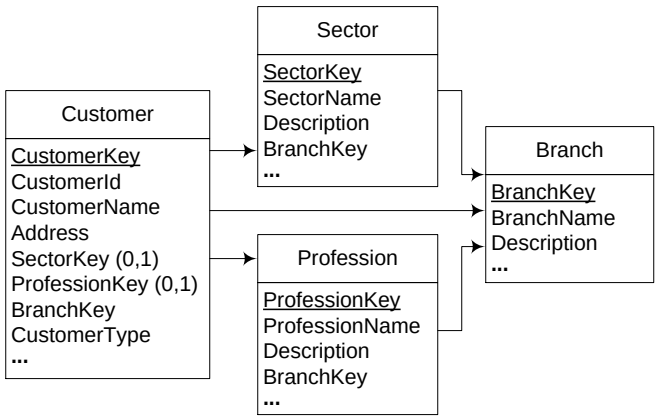


Fig. 5.8. Improved relational representation of the generalized hierarchy in Fig. 4.4

An example of the relations for the hierarchy in Fig. 4.4 is given in Fig. 5.8. The table Customer includes two kinds of foreign keys: one that indicates the next specialized hierarchy level (SectorKey and ProfessionKey), which is obtained by applying Rules 1 and 3b in Sect. 5.3; the other kind of foreign key corresponds to the next joining level (BranchKey), which is obtained by applying Rule 4 above. The discriminating attribute CustomerType, which can take the values Person and Company, indicates the specific aggregation path of members to facilitate aggregations. Finally, **check constraints** must be

specified to ensure that only one of the foreign keys for the specialized levels may have a value, according to the value of the discriminating attribute.

```
ALTER TABLE Customer ADD CONSTRAINT CustomerTypeCK
        CHECK ( CustomerType IN ('Person', 'Company') )
ALTER TABLE Customer ADD CONSTRAINT CustomerPersonFK
        CHECK ( (CustomerType != 'Person') OR
        ( ProfessionKey IS NOT NULL AND SectorKey IS NULL ) )
ALTER TABLE Customer ADD CONSTRAINT CustomerCompanyFK
        CHECK ( (CustomerType != 'Company') OR
        ( ProfessionKey IS NULL AND SectorKey IS NOT NULL ) )
```

The schema in Fig. 5.8 allows one to choose alternative paths for analysis. One possibility is to use the paths that include the specific levels, for example Profession or Sector. Another possibility is to only access the levels that are common to all members, for example, to analyze all customers, whatever their type, using the hierarchy Customer and Branch. As with the snowflake structure, one disadvantage of this structure is the necessity to apply join operations between several tables. However, an important advantage is the expansion of the analysis possibilities that it offers.

The mapping above can also be applied to **ragged hierarchies** since these hierarchies are a special case of generalized hierarchies. This is illustrated in Fig. 5.4 where the City level has two foreign keys to the State and Country levels. Nevertheless, since in a ragged hierarchy there is a unique path where some levels can be skipped, another solution is to embed the attributes of an optional level in the splitting level. This is illustrated in Fig. 5.4, where the level State has two optional attributes corresponding to the Region level. Finally, another solution would be to transform the hierarchy at the instance level by including placeholders in the missing intermediate levels, as it is done for unbalanced hierarchies in Sect. 5.5.2. In this way, a ragged hierarchy is converted into a balanced hierarchy and a star or snowflake structure can be used for its logical representation.

5.5.4 Alternative Hierarchies

For alternative hierarchies, the traditional mapping to relational tables can be applied. This is shown in Fig. 5.9 for the conceptual schema in Fig. 4.7. Note that even though generalized and alternative hierarchies can be easily distinguished at the conceptual level (see Figs. 4.4a and 4.7), this distinction cannot be made at the logical level (compare Figs. 5.7 and 5.9).

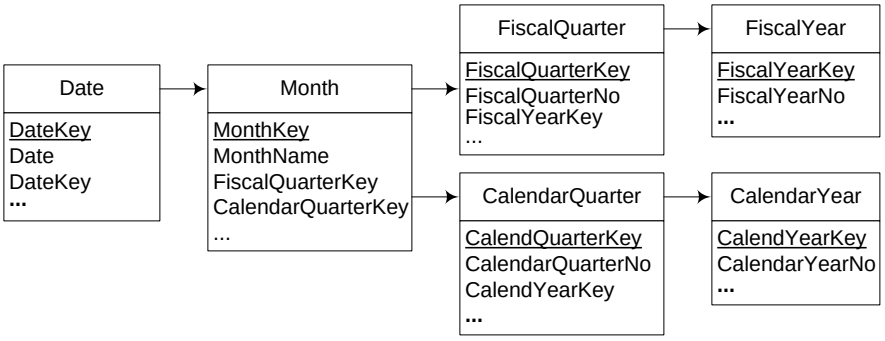


Fig. 5.9. Relations for the alternative hierarchy in Fig. 4.7

5.5.5 Parallel Hierarchies

As parallel hierarchies are composed of several hierarchies, their logical mapping consists in combining the mappings for the specific types of hierarchies. For example, Fig. 5.10 shows the result of applying this mapping to the schema shown in Fig. 4.9.

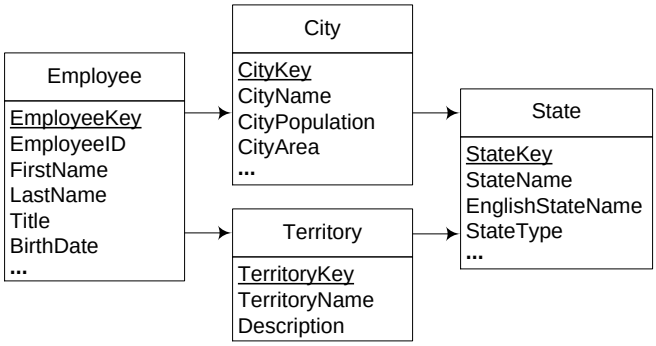


Fig. 5.10. Relations for the parallel dependent hierarchies in Fig. 4.10

Note that shared levels in parallel dependent hierarchies are represented in one table (State, in this example). Since these levels play different roles in each hierarchy, we can create views in order to facilitate queries and visualization. For example, in Fig. 5.10 table States contains all states where an employee lives, works, or both. Therefore, aggregating along the path Employee → City → State will yield states where no employee lives. If we do not want these states in the result, we can create a view named StateLives containing only the states where at least one employee lives.

Finally, note also that both alternative and parallel dependent hierarchies can be easily distinguished at the conceptual level (Figs. 4.7 and 4.10); however, their logical representations (Figs. 5.9 and 5.10) look similar in spite of several characteristics that differentiate them, as explained in Sect. 4.2.5.

5.5.6 Nonstrict Hierarchies

Applying the mapping rules specified in Sect. 5.3 to nonstrict hierarchies, creates relations for the levels and an additional relation (called a **bridge table**) for the many-to-many relationship between them. An example for the hierarchy in Fig. 4.13 is given in Fig. 5.11, where the bridge table EmplSection represents the many-to-many relationship. If the parent-child relationship has a distributing attribute (as in Fig. 4.13), the bridge table will include an additional attribute for storing the values required for measure distribution. However, in order to aggregate measures correctly, a special aggregation procedure that uses this distributing attribute must be implemented.

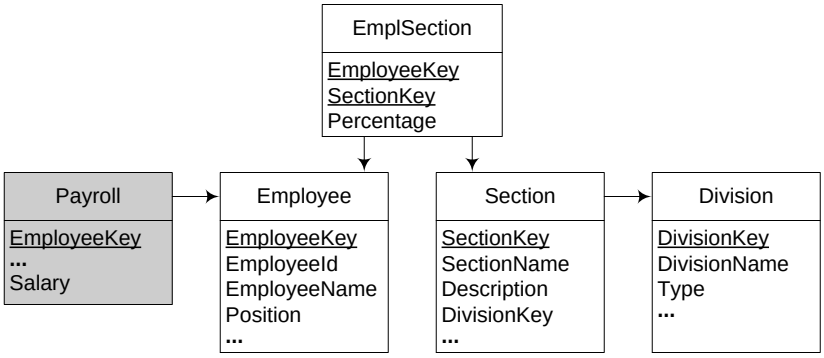


Fig. 5.11. Relations for the nonstrict hierarchy in Fig. 4.13

Recall from Sect. 4.2.6 that another solution is to transform a nonstrict hierarchy into a strict one by including an additional dimension in the fact, as shown in Fig. 4.14. Then, the corresponding mapping for a strict hierarchy can be applied. The choice between the two solutions may depend on various factors, namely,

- Data structure and size: Bridge tables require less space than creating additional dimensions. In the latter case, the fact table grows if child members are related to many parent members. The additional foreign key in the fact table also increases the space required. In addition, for bridge tables, information about the parent-child relationship and distributing attribute (if it exists) must be stored separately.

- Performance and applications: For bridge tables, join operations, calculations, and programming effort are needed to aggregate measures correctly, while in the case of additional dimensions, measures in the fact table are ready for aggregation along the hierarchy. Bridge tables are thus appropriate for applications that have a few nonstrict hierarchies. They are also adequate when the information about measure distribution does not change with time. On the contrary, additional dimensions can easily represent changes in time of measure distribution.

Finally, still another option consists in transforming the many-to-many relationship into a one-to-many relationship by defining a "primary" relationship, that is, to convert the nonstrict hierarchy into a strict one, to which the corresponding mapping for simple hierarchies is applied (as explained in Sect. 4.3.2).

5.6 Advanced Modeling Aspects

We discuss next how facts with multiple granularities, many-to-many dimensions, and links between facts can be represented in the relational model.

5.6.1 Facts with Multiple Granularities

Two approaches can be used for the logical representation of facts with multiple granularities. The first one consists in using multiple foreign keys, one for each alternative granularity, in a similar way as it was explained for generalized hierarchies in Sect. 5.5.3. The second approach consists in removing granularity variation at the instance level with the help of placeholders, in a similar way as explained for unbalanced hierarchies in Sect. 5.5.2.

Consider the example of Fig. 4.16, where measures are registered at multiple granularities. Figure 5.12 shows the relational schema resulting from the first solution above, where the Sales fact table is related to both the City and the State levels through referential integrity constraints. In this case, both attributes CityKey and StateKey are optional, and constraints must be specified to ensure that only one of the foreign keys may have a value.

Figure 5.13 shows an example of instances for the second solution above, where placeholders are used for facts that refer to nonleaf levels. There are two possible cases illustrated by the two placeholders in the figure. In the first case, a fact member points to a nonleaf member that has children. In this case, placeholder PH1 represents all cities other than the existing children. In the second case, a fact member points to a nonleaf member without children. In this case, placeholder PH2 represents all (unknown) cities of the state.

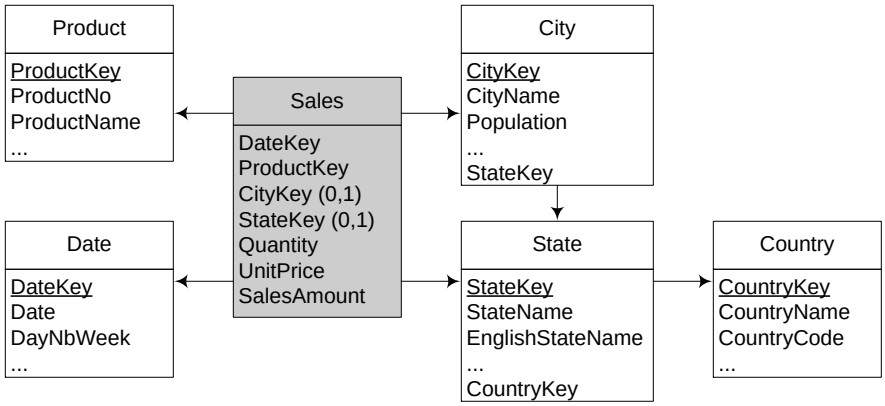


Fig. 5.12. Relations for the fact with multiple granularities in Fig. 4.16

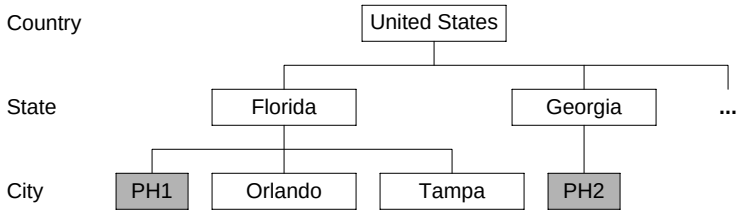


Fig. 5.13. Using placeholders for the fact with multiple granularities in Fig. 4.16

Obviously, in both solutions, the issue is to guarantee the correct summarization of measures. In the first solution, when aggregating at the state level, we need to perform a union of two subqueries, one for each alternative path. In the second solution, when aggregating at the city level, we obtain the placeholders in the result.

5.6.2 Many-to-Many Dimensions

The mapping to the relational model given in Sect. 5.3, applied to many-to-many dimensions, creates relations representing the fact, the dimension levels, and an additional bridge table representing the many-to-many relationship between the fact table and the dimension. Figure 5.14 shows the relational representation of the many-to-many dimension in Fig. 4.17. As can be seen, a bridge table BalanceClient relates the fact table Balance with the dimension table Client. Note also that a surrogate key was added to the Balance fact table so it can be used in the bridge table for relating facts with clients.

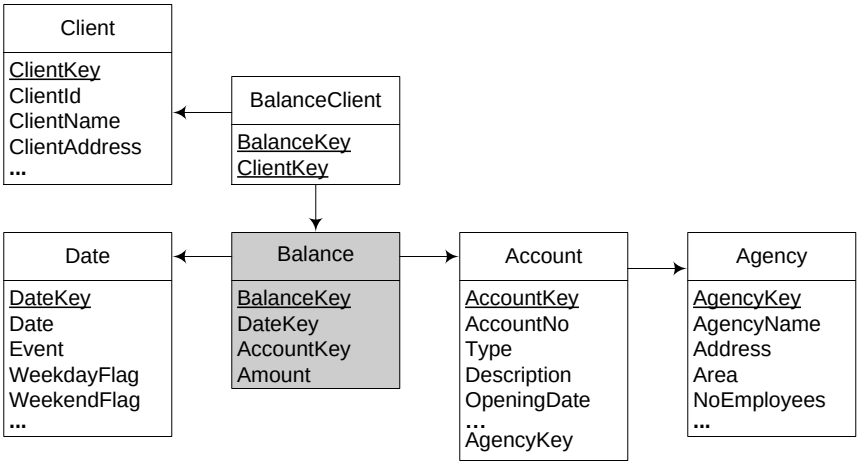


Fig. 5.14. Relations for the many-to-many dimension in Fig. 4.17

We have seen in Sect. 4.3.2 several solutions to decompose a many-to-many dimension according to the dependencies that hold on the fact table. In this case, after the decomposition, the traditional mapping to the relational model can be applied to the resulting decomposition.

5.6.3 Links between Facts

The logical representation of links between facts depends on their cardinalities. In the case of one-to-one or one-to-many cardinalities, the surrogate key of the fact with a one cardinality is added as a foreign key of the other fact. In the case of many-to-many cardinalities, a bridge table with foreign keys to the two facts is needed.

Consider the example of Fig. 4.21, where there is a many-to-many link between the Order and Delivery facts. Figure 5.15 shows the corresponding relational schema, where both facts have surrogates keys and the bridge table OrderDelivery relates the two facts. On the other hand, suppose that the link between the facts is one-to-many, that is, an order is delivered by only one delivery, while a delivery may concern several orders. In this case, a bridge table is no longer needed and the DeliveryKey should be added to the Order fact table.

The link between fact tables is commonly used for combining the data from two different cubes through a join operation. In our example, this would combine information about orders and deliveries in order to analyze the entire sales process. A join between the two fact tables through the bridge table would have the following schema

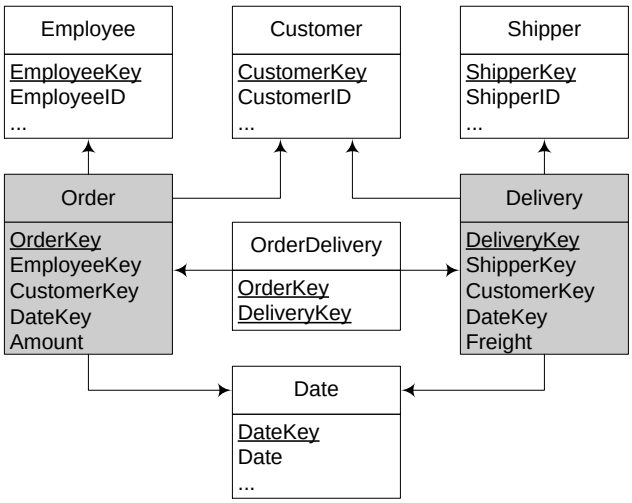


Fig. 5.15. Relations for the schema with a link between facts in Fig. 4.21

OrderDelivery(OrderKey, DeliveryKey, EmployeeKey, ShipperKey, CustomerKey, OrderDateKey, DeliveryDateKey, Amount, Freight)

Notice that only one copy of the CustomerKey is kept (since it is assumed that it is the same in both orders and deliveries), while the DateKey of both facts are kept, and these are renamed OrderDateKey and DeliveryDateKey. Notice, that since the relationship between the two facts is many-to-many, this may induce the double counting problem to which we referred in Sect. 4.2.6.

5.7 Slowly Changing Dimensions

So far, we have assumed that new data that arrives to the warehouse only corresponds to facts, which means dimensions are stable, and their data do not change. However, in many real-world situations, dimensions can change both at the structure and the instance level. Structural changes occur, for example, when an attribute is deleted from the data sources and therefore it is no longer available. As a consequence, this attribute should also be deleted from the dimension table. Changes at the instance level can be of two kinds. First, when a correction must be made to the dimension tables due to an error, the new data should replace the old one. Second, when the contextual conditions of an analysis scenario change, the contents of dimension tables must change accordingly. We cover these two latter cases in this section.

We will introduce the problem by means of a simplified version of the Northwind data warehouse. In this simplified version, we consider a Sales fact

table related only to the dimensions Date, Employee, Customer, and Product, and a SalesAmount measure. We assume a star (denormalized) representation of table Product, and thus category data are embedded in this table. Below, we show instances of the Sales fact table and the Product dimension table.

DateKey	EmployeeKey	CustomerKey	ProductKey	SalesAmount
t1	e1	c1	p1	100
t2	e2	c2	p1	100
t3	e1	c3	p3	100
t4	e2	c4	p4	100

ProductKey	ProductName	UnitPrice	CategoryName	Description
p1	prod1	10.00	cat1	desc1
p2	prod2	12.00	cat1	desc1
p3	prod3	13.50	cat2	desc2
p4	prod4	15.00	cat2	desc2

As we said above, new tuples will be entered into the Sales fact table as new sales occur. But also other updates are likely to occur. For example, when a new product starts to be commercialized by the company, a new tuple in Product must be inserted. Also, data about a product may be wrong, and in this case, the corresponding tuples must be corrected. Finally, the category of a product may change as a result of a new commercial or administrative policy. Assuming that these kinds of changes are not frequent, when the dimensions are designed so that they support them, they are called *slowly changing dimensions*.

In the scenario above, consider a query asking for the total sales per employee and product category, expressed as follows:

```
SELECT    E.EmployeeKey, P.CategoryName, SUM(SalesAmount)
FROM      Sales S, Product P
WHERE     S.ProductKey = P.ProductKey
GROUP BY E.EmployeeKey, P.CategoryName
```

This query would return the following table:

EmployeeKey	CategoryName	SalesAmount
e1	cat1	100
e2	cat1	100
e1	cat2	100
e2	cat2	100

Suppose now that, at an instant t after t4 (the date of the last sale shown in the fact table above), the category of product prod1 changed to cat2, that means, there is a reclassification of the product with respect to its category. The trivial solution of updating the category of the product to cat2 will, in general, produce erroneous results since there is no track of the previous category of a product. For example, if the user poses the same query as above, and the fact table has not been changed in the meantime, she would expect to

get the same result, but since all the sales occurred before the reclassification, she would get the following result:

EmployeeKey	CategoryName	SalesAmount
e1	cat2	200
e2	cat2	200

This result is incorrect since the products affected by the category change were already associated with sales data. Opposite to this, if the new category would be the result of an error correction (i.e., the actual category of prod1 is cat2), this result would be correct. In the former case, obtaining the correct answer requires to guarantee the preservation of the results obtained when prod1 had category cat1, and make sure that the new aggregations will be computed with the new category cat2.

Three basic ways of handling slowly changing dimensions have been proposed in the literature. The simplest one, called type 1, consists in overwriting the old value of the attribute with the new one, which implies that we lose the history of the attribute. This approach is appropriate when the modification is due to an error in the dimension data.

In the second solution, called type 2, the tuples in the dimension table are versioned, and a new tuple is inserted each time a change takes place. In our example, we would enter a new row for product prod1 in the Product table with its new category cat2 so that all sales prior to t will contribute to the aggregation to cat1, while the ones that occurred after t will contribute to cat2. This solution requires to have a surrogate key in the dimension in addition to the business key so that all versions of a dimension member have different surrogate key but the same business key. In our example, we keep the business key in a column ProductID and the surrogate key in a column ProductKey. Furthermore, it also necessary to extend the table Product with two attributes indicating the validity interval of the tuple, let us call them From and To. The table Product would look like the following:

Product Key	Product ID	Product Name	UnitPrice	Category Name	Description	From	To
k1	p1	prod1	10.00	cat1	desc1	2010-01-01	2011-12-31
k11	p1	prod1	10.00	**cat2**	desc2	2012-01-01	9999-12-31
k2	p2	prod2	12.00	cat1	desc1	2010-01-01	9999-12-31
k3	p3	prod3	13.50	cat2	desc2	2010-01-01	9999-12-31
k4	p4	prod4	15.00	cat2	desc2	2011-01-01	9999-12-31

In the table above, the first two tuples correspond to the two versions of product prod1, with ProductKey values k1 and k11. The value 9999-12-31 in the To attribute indicates that the tuple is still valid; this is a usual notation in temporal databases. Since the same product participates in the fact table with as many surrogates as there are attribute changes, the business key keeps track of all the tuples that pertain to the same product. For example, the business key will be used when counting the number of different products sold by the company over specific time periods. Notice that since a new

record is inserted every time an attribute value changes, the dimension can grow considerably, decreasing the performance during join operations with the fact table. More sophisticated techniques have been proposed to address this, and below we will comment on them.

In the type 2 approach, sometimes an additional attribute is added to explicitly indicate which is the current row. The table below shows an attribute denoted RowStatus, telling which is the current value for product prod1.

Product Key	Product ID	Product Name	Unit Price	Category Name	Description	From	To	Row Status
k1	p1	prod1	10.00	cat1	desc1	2010-01-01	2011-12-31	Expired
k11	p1	prod1	10.00	**cat2**	desc2	2012-01-01	9999-12-31	Current
...	...	...	...	...	...	...	...	...

Let us consider a snowflake representation for the Product dimension where the categories are represented in a table Category, as given next.

Product Key	Product Name	Unit Price	Category Key
p1	prod1	10.00	c1
p2	prod2	12.00	c1
p3	prod3	13.50	c2
p4	prod4	15.00	c2

Category Key	Category Name	Description
c1	cat1	desc1
c2	cat2	desc2
c3	cat3	desc3
c4	cat4	desc4

The type 2 approach for a snowflake representation is handled in similar way as above, but now a surrogate key and two temporal attributes From and To must be added to both the Product and the Category tables. Assume that, as before, product prod1 changes its category to cat2. In the case of a solution of type 2, the Product table will be as shown below

Product Key	Product ID	Product Name	Unit Price	Category Key	From	To
k1	p1	prod1	10.00	l1	2010-01-01	2011-12-31
k11	p1	prod1	10.00	**l2**	2012-01-01	9999-12-31
k2	p2	prod2	12.00	l1	2010-01-01	9999-12-31
k3	p3	prod3	13.50	l2	2010-01-01	9999-12-31
k4	p4	prod4	15.00	l2	2011-01-01	9999-12-31

and the Category table remains unchanged. However, if the change occurs at an upper level in the hierarchy, this change needs to be propagated downward in the hierarchy. For example, suppose that the description of category cat1 changes, as reflected in the following table:

Category Key	Category ID	Category Name	Description	From	To
l1	c1	cat1	desc1	2010-01-01	2011-12-31
l11	c1	cat1	**desc11**	2012-01-01	9999-12-31
l2	c2	cat2	desc2	2010-01-01	9999-12-31
l3	c3	cat3	desc3	2010-01-01	9999-12-31
l4	c4	cat4	desc4	2010-01-01	9999-12-31

This change must be propagated to the Product table so that all sales prior to the change refer to the old version of category cat1 (with key l1), while the new sales must point to the new version (with key l11), as shown below:

Product Key	Product ID	Product Name	Unit Price	Category Key	From	To
k1	p1	prod1	10.00	l1	2010-01-01	2011-12-31
k11	p1	prod1	10.00	**l11**	2012-01-01	9999-12-31
k2	p2	prod2	12.00	l1	2010-01-01	2011-12-31
k21	p2	prod2	12.00	**l11**	2012-01-01	9999-12-31
k3	p3	prod3	13.50	l2	2010-01-01	9999-12-31
k4	p4	prod4	15.00	l2	2011-01-01	9999-12-31

The third solution to the problem of slowly changing dimensions, called type 3, consists in introducing an additional column for each attribute subject to change, which will hold the new value of the attribute. In our case, attributes CategoryName and Description changed since when product prod1 changes category from cat1 to cat2, the associated description of the category also changes from desc1 to desc2. The following table illustrates this solution:

Product Key	Product Name	UnitPrice	Category Name	New Category	Description	New Description
p1	prod1	10.00	**cat1**	**cat2**	**desc1**	**desc2**
p2	prod2	12.00	cat1		desc1	
p3	prod3	13.50	cat2		desc2	
p4	prod4	15.00	cat2		desc2	

Note that only the two last versions of the attribute can be represented in this solution and that the validity interval of the tuples is not stored.

It is worth noticing that it is possible to apply the three solutions above, or combinations of them, to the same dimension. For example, we may apply correction (type 1), tuple versioning (type 2), or attribute addition (type 3) for various attributes in the same dimension table.

In addition to these three classic approaches to handle slowly changing dimensions, more sophisticated (although more difficult to implement) solutions have been proposed. We briefly comment on them next.

The type 4 approach aims at handling very large dimension tables and attributes that change frequently. This situation can make the dimension tables to grow to a point that even browsing the dimension can become very slow. Thus, a new dimension, called a **minidimension**, is created to store the most frequently changing attributes. For example, assume that in the Product dimension there are attributes SalesRanking and PriceRange, which are likely to change frequently, depending on the market conditions. Thus, we will create a new dimension called ProductFeatures, with key ProductFeaturesKey, and the attributes SalesRanking and PriceRange, as follows:

Product FeaturesKey	Sales Ranking	Price Range
pf1	1	1-100
pf2	2	1-100
...	...	...
pf200	7	500-600

As can be seen, there will be one row in the minidimension for each unique combination of SalesRanking and PriceRange encountered in the data, not one row per product.

The key ProductFeaturesKey must be added to the fact table Sales as a foreign key. In this way, we prevent the dimension to grow with every change in the sales ranking score or price range of a product, and the changes are actually captured by the fact table. For example, assume that product prod1 initially has sales ranking 2 and price range 1-100. A sale of this product will be entered in the fact table with a value of ProductFeaturesKey equal to pf2. If later the sales ranking of the product goes up to 1, the subsequent sales will be entered with ProductFeaturesKey equal to pf1.

The type 5 approach is an extension of type 4, where the primary dimension table is extended with a foreign key to the minidimension table. In the current example, the Product dimension will look as follows:

Product Key	Product Name	UnitPrice	CurrentProduct FeaturesKey
p1	prod1	10.00	pf1
...	...	...	...

As can be seen, this allows us to analyze the current feature values of a dimension without accessing the fact table. The foreign key is a type 1 attribute, and thus, when any feature of the product changes, the current ProductFeaturesKey value is stored in the Product table. On the other hand, the fact table includes the foreign keys ProductKey and ProductFeaturesKey, where the latter points to feature values that were current *at the time of the sales*. However, the attribute CurrentProductFeaturesKey in the Product dimension would allow us to roll up historical facts based on the current product profile.

The type 6 approach extends a type 2 dimension with an additional column containing the current value of an attribute. Consider again the type 2 solution above for the snowflake representation, where the Product dimension has a surrogate key ProductKey, a business key ProductID, and attributes From and To indicating the validity interval of the tuple. Further, we add an attribute CurrentCategoryKey that contains the current value of the Category attribute as follows:

Product Key	Product ID	Product Name	Unit Price	Category Key	From	To	Current CategoryKey
k1	p1	prod1	10.00	l1	2010-01-01	2011-12-31	**l11**
k11	p1	prod1	10.00	**l11**	2012-01-01	9999-12-31	**l11**
k2	p2	prod2	12.00	l1	2010-01-01	9999-12-31	l1
k3	p3	prod3	13.50	l2	2010-01-01	9999-12-31	l2
k4	p4	prod4	15.00	l2	2011-01-01	9999-12-31	l2

In this case, the CategoryKey attribute can be used to group facts based on the category that was in effect when the facts occurred, while the CurrentCategoryKey attribute can be used to group facts based on the current category.

Finally, the type 7 approach delivers similar functionality as the type 6 solution in the case that there are many attributes in the dimension table for which we need to support both current and historical perspectives. In a type 6 solution that would require one additional column in the dimension table for each of such attributes, these columns will contain the current value of the attributes. Instead, a type 7 solution would add to the fact table an additional foreign key of the dimension table containing the natural key (ProductID in our example), provided it is a *durable* one. In our example, the Product dimension will be exactly the same as in the type 2 solution, but the fact table would look as follows:

DateKey	EmployeeKey	CustomerKey	ProductKey	ProductID	SalesAmount
t1	e1	c1	k1	p1	100
t2	e2	c2	k11	p1	100
t3	e1	c3	k3	p3	100
t4	e2	c4	k4	p4	100

The ProductKey column can be used for historical analysis based on the product values effective when the fact occurred, while the ProductID column will be used for current values. In order to support current analysis we need an additional view, called CurrentProduct, which keeps only current values of the Product dimension as follows:

ProductID	ProductName	UnitPrice	CategoryKey
p1	prod1	10.00	l2
p2	prod2	12.00	l1
p3	prod3	13.50	l2
p4	prod4	15.00	l2

A variant of this approach uses the surrogate key as the key of the current dimension, thus eliminating the need of handling two different foreign keys in the fact table.

Leading data warehouse platforms provide some support for slowly changing dimensions, typically type 1 to type 3. However, as we have seen, the proposed solutions are not satisfactory. In particular, they require considerable programming effort for their correct manipulation. As we will discuss in Chap. **??**, temporal data warehouses have been proposed as a more general solution to this problem. They aim at providing built-in temporal semantics to data warehouses.

5.8 SQL/OLAP

In this section, we briefly introduce an extension of SQL, called SQL/OLAP, that provides functionality for analytical queries.

A relational database is not the best data structure to hold multidimensional data. Consider a simple cube Sales, with two dimensions Product and Customer, and a measure SalesAmount, as depicted in Fig. 5.16a. This data cube contains all possible (2^2) aggregations of the cube cells, namely, SalesAmount by Product, by Customer, and by both Product and Customer, in addition to the base nonaggregated data.

	c1	c2	c3	Total
p1	100	105	100	305
p2	70	60	40	170
p3	30	40	50	120
Total	200	205	190	595

(a)

ProductKey	CustomerKey	SalesAmount
p1	c1	100
p1	c2	105
p1	c3	100
p2	c1	70
p2	c2	60
p2	c3	40
p3	c1	30
p3	c2	40
p3	c3	50

(b)

Fig. 5.16. (**a**) A data cube with two dimensions, Product and Customer. (**b**) A fact table representing the same data

Figure 5.16b shows a relational fact table corresponding to Fig. 5.16a. Computing all aggregations of a cube with n dimensions would require 2^n GROUP BY statements, which is not very efficient. For this reason, SQL/OLAP extends the GROUP BY clause with the ROLLUP and CUBE operators. The former computes group subtotals in the order given by a list of attributes. The latter computes all totals of such a list. Over the grouped tuples, the HAVING clause can be applied, as in a typical GROUP BY.

The syntax of both statements applied to our example above are

```
SELECT    ProductKey, CustomerKey, SUM(SalesAmount)
FROM      Sales
GROUP BY ROLLUP(ProductKey, CustomerKey)

SELECT    ProductKey, CustomerKey, SUM(SalesAmount)
FROM      Sales
GROUP BY CUBE(ProductKey, CustomerKey)
```

Figure 5.17a,b show, respectively, the result of the GROUP BY ROLLUP and the GROUP BY CUBE queries above. In the case of roll-up, in addition

to the detailed data, we can see the total amount by product and the overall total. For example, the total sales for product p1 is 305. If we also need the totals by customer, we would need the cube computation, performed by the second query.

ProductKey	CustomerKey	SalesAmount
p1	c1	100
p1	c2	105
p1	c3	100
p1	NULL	305
p2	c1	70
p2	c2	60
p2	c3	40
p2	NULL	170
p3	c1	30
p3	c2	40
p3	c3	50
p3	NULL	120
NULL	NULL	595

(a)

ProductKey	CustomerKey	SalesAmount
p1	c1	100
p2	c1	70
p3	c1	30
NULL	c1	200
p1	c2	105
p2	c2	60
p3	c2	40
NULL	c2	205
p1	c3	100
p2	c3	40
p3	c3	50
NULL	c3	190
p1	NULL	305
p2	NULL	170
p3	NULL	120
NULL	NULL	595

(b)

Fig. 5.17. Operators (**a**) GROUP BY ROLLUP and (**b**) GROUP BY CUBE

Actually, the ROLLUP and CUBE operators are simply shorthands for a more powerful operator, called GROUPING SETS, which is used to precisely specify the aggregations to be computed. For example, the GROUP BY ROLLUP query above can be written using GROUPING SETS as follows:

```
SELECT    ProductKey, CustomerKey, SUM(SalesAmount)
FROM      Sales
GROUP BY GROUPING SETS((ProductKey, CustomerKey), (ProductKey), ())
```

Analogously, the GROUP BY CUBE query would read:

```
SELECT    ProductKey, CustomerKey, SUM(SalesAmount)
FROM      Sales
GROUP BY GROUPING SETS((ProductKey, CustomerKey),
          (ProductKey), (CustomerKey), ())
```

A very common OLAP need is to compare detailed data with aggregate values. For example, we may need to compare the sales of a product to a customer against the maximum sales of this product to any customer. Thus, we could obtain the relevance of each customer with respect to the sales of the

product. SQL/OLAP provides the means to perform this through a feature called **window partitioning**. This query would be written as follows:

```
SELECT ProductKey, CustomerKey, SalesAmount, MAX(SalesAmount) OVER
       (PARTITION BY ProductKey) AS MaxAmount
FROM   Sales
```

The result of the query is given in Fig. 5.18. The first three columns are obtained from the initial Sales table. The fourth one is obtained as follows. For each tuple, a window is defined, called *partition*, containing all the tuples pertaining to the same product. The attribute SalesAmount is then aggregated over this group using the corresponding function (in this case MAX) and the result is written in the MaxAmount column. Note that the first three tuples, corresponding to product p1, have a MaxAmount of 105, that is, the maximum amount sold of this product to customer c2.

ProductKey	CustomerKey	SalesAmount	MaxAmount
p1	c1	100	105
p1	c2	105	105
p1	c3	100	105
p2	c1	70	70
p2	c2	60	70
p2	c3	40	70
p3	c1	30	50
p3	c2	40	50
p3	c3	50	50

Fig. 5.18. Sales of products to customers compared with the maximum amount sold for that product

A second SQL/OLAP feature, called **window ordering**, is used to order the rows within a partition. This feature is useful, in particular, to compute rankings. Two common aggregate functions applied in this respect are ROW_NUMBER and RANK. For example, the next query shows how does each product rank in the sales of each customer. For this, we can partition the table by customer, and apply the ROW_NUMBER function as follows:

```
SELECT ProductKey, CustomerKey, SalesAmount, ROW_NUMBER() OVER
       (PARTITION BY CustomerKey ORDER BY SalesAmount DESC) AS RowNo
FROM   Sales
```

The result is shown in Fig. 5.19a. The first tuple, for example, was evaluated by opening a window with all the tuples of customer c1, ordered by the sales amount. We see that product p1 is the one most demanded by customer c1.

We could instead partition by product, and study how each customer ranks in the sales of each product, using the function RANK.

Product Key	Customer Key	Sales Amount	RowNo
p1	c1	100	1
p2	c1	70	2
p3	c1	30	3
p1	c2	105	1
p2	c2	60	2
p3	c2	40	3
p1	c3	100	1
p3	c3	50	2
p2	c3	40	3

(a)

Product Key	Customer Key	Sales Amount	Rank
p1	c2	105	1
p1	c3	100	2
p1	c1	100	2
p2	c1	70	1
p2	c2	60	2
p2	c3	40	3
p3	c3	50	1
p3	c2	40	2
p3	c1	30	3

(b)

Fig. 5.19. (**a**) Ranking of products in the sales of customers; (**b**) Ranking of customers in the sales of products

```
SELECT ProductKey, CustomerKey, SalesAmount, RANK() OVER
       (PARTITION BY ProductKey ORDER BY SalesAmount DESC) AS Rank
FROM   Sales
```

As shown in the result given in Fig. 5.19b, the first tuple was evaluated opening a window with all the tuples with product p1, ordered by the sales amount. We can see that customer c2 is the one with highest purchases of p1, and customers c3 and c1 are in the second place, with the same ranking.

A third kind of feature of SQL/OLAP is **window framing**, which defines the size of the partition. This is used to compute statistical functions over time series, like moving averages. To give an example, let us assume that we add two columns Year and Month to the Sales table. The following query computes the 3-month moving average of sales by product.

```
SELECT ProductKey, Year, Month, SalesAmount, AVG(SalesAmount) OVER
       (PARTITION BY ProductKey ORDER BY Year, Month
       ROWS 2 PRECEDING) AS MovAvg
FROM   Sales
```

The result is shown in Fig. 5.20a. For each tuple, the query evaluator opens a window that contains the tuples pertaining to the current product. Then, it orders the window by year and month and computes the average over the current tuple and the preceding two ones, provided they exist. For example, in the first tuple, the average is computed over the current tuple (there is no preceding tuple), while in the second tuple, the average is computed over the current tuple and the preceding one. Finally, in the third tuple, the average is computed over the current tuple and the two preceding ones.

As another example, the following query computes the year-to-date sum of sales by product.

```
SELECT ProductKey, Year, Month, SalesAmount, AVG(SalesAmount) OVER
```

Product Key	Year	Month	Sales Amount	MovAvg
p1	2011	10	100	100
p1	2011	11	105	102.5
p1	2011	12	100	101.67
p2	2011	12	60	60
p2	2012	1	40	50
p2	2012	2	70	56.67
p3	2012	1	30	30
p3	2012	2	50	40
p3	2012	3	40	40

(a)

Product Key	Year	Month	Sales Amount	YTD
p1	2011	10	100	100
p1	2011	11	105	205
p1	2011	12	100	305
p2	2011	12	60	60
p2	2012	1	40	40
p2	2012	2	70	110
p3	2012	1	30	30
p3	2012	2	50	80
p3	2012	3	40	120

(b)

Fig. 5.20. (**a**) Three-month moving average of the sales per product (**b**) Year-to-date sum of the sales per product

```
        (PARTITION BY ProductKey, Year ORDER BY Month
        ROWS UNBOUNDED PRECEDING) AS YTD
FROM    Sales
```

The result is shown in Fig. 5.20b. For each tuple, the query evaluator opens a window that contains the tuples pertaining to the current product and year ordered by month. Unlike in the previous query, the aggregation function SUM is applied to all the tuples before the current tuple, as indicated by ROWS UNBOUNDED PRECEDING.

It is worth noting that queries that use window functions can be expressed without them, although the resulting queries are harder to read and may be less efficient. For example, the query above computing the year-to-date sales can be equivalently written as follows:

```
SELECT ProductKey, Year, Month, SalesAmount, AVG(SalesAmount) AS YTD
FROM   Sales S1, Sales S2
WHERE  S1.ProductKey = S2.ProductKey AND
       S1.Year = S2.Year AND S1.Month >= S2.Month
```

Of course, there are many other functions provided in the SQL/OLAP extension, which the interested reader can find in the standard.

5.9 Definition of the Northwind Cube in Analysis Services

We introduce next the main concepts of Analysis Services using as example the Northwind cube. In this section, we consider a simplified version of the

Northwind cube where the ragged geography hierarchy was transformed into a regular one. The reason for this was to simplify both the schema definition and the associated MDX and SQL queries that we will show in the next chapter. More precisely, we did not include sales data about cities that roll-up to the country level, such as Singapore. Therefore, we dropped the foreign key CountryKey in table City. Moreover, we did not consider the Region level. As a result, the hierarchy City → State → Country → Continent becomes balanced.

To define a cube in Analysis Services, we use SQL Server Data Tools introduced in Chap. **??**. The various kinds of objects to be created are described in detail in the remainder of this section.

5.9.1 Data Sources

A data warehouse retrieves its data from one or several data stores. A **data source** contains connection information to a data store, which includes the location of the server, a login and password, a method to retrieve the data, and security permissions. Analysis Services supports data sources that have a connectivity interface through OLE DB or .NET Managed Provider. If the source is a relational database, then SQL is used by default to query the database. In our example, there is a single data source that connects to the Northwind data warehouse.

5.9.2 Data Source Views

A **data source view** (DSV) defines the relational schema that is used for populating an Analysis Services database. This schema is derived from the schemas of the various data sources. Indeed, some transformations are often needed in order to load data from sources into the warehouse. For example, common requirements are to select some columns from a table, to add a new derived column to a table, to restrict table rows on the basis of some specific criteria, and to merge several columns into a single one. These operations can be performed in the DSV by replacing a source table with a **named query** written in SQL or by defining a **named calculation**, which adds a derived column defined by an SQL expression. Further, if the source systems do not specify the primary keys and the relationships between tables using foreign keys, these can be defined in the DSV.

Analysis Services allows the user to specify friendly names for tables and columns. In order to facilitate visibility and navigation for large data warehouses, it also offers the possibility to define customizable views within a DSV, called **diagrams**, that show only certain tables.

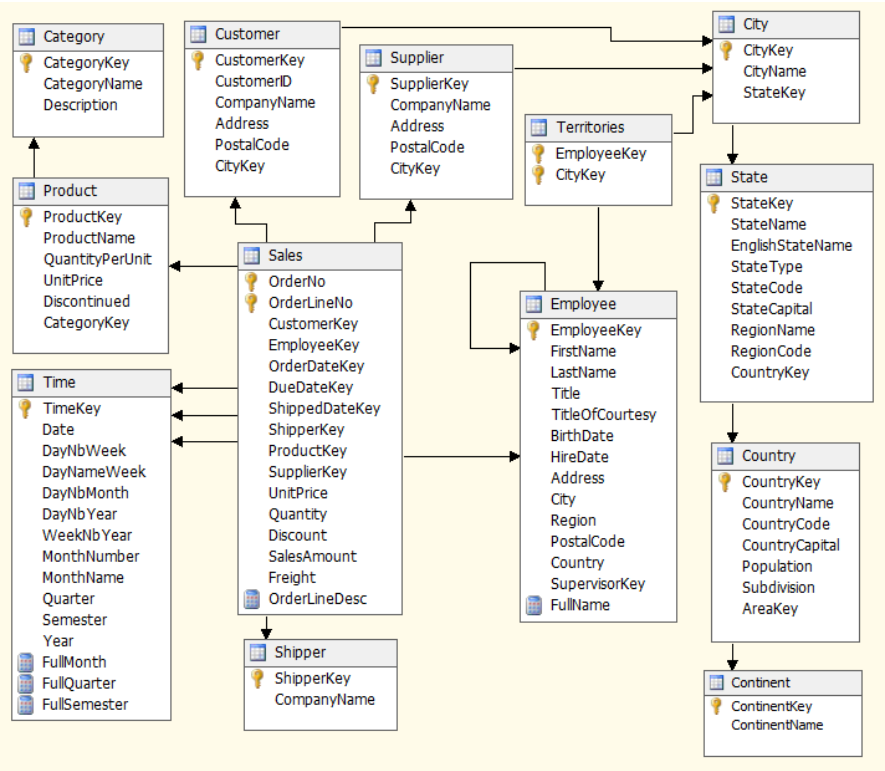


Fig. 5.21. The data source view for the Northwind cube

The DSV, based on the Northwind data warehouse of Fig. 5.4 is given in Fig. 5.21. We can see the Sales fact table and the associated dimension tables (recall that the ragged geography hierarchy was transformed into a regular one). The figure also shows several named calculations, which are identified by a special icon at the left of the attribute name. As we will see later, these named calculations are used for defining and browsing the dimensions. The calculations are:

- In the Employee dimension table, the named calculation FullName combines the first and last name with the expression

  FirstName + ' ' + LastName

- In the Date dimension table, the named calculations FullMonth, FullQuarter, and FullSemester, are defined, respectively, by the expressions

  MonthName + ' ' + CONVERT(CHAR(4),Year)
  'Q' + CONVERT(CHAR(1), Quarter) + ' ' + CONVERT(CHAR(4), Year)
  'S' + CONVERT(CHAR(1), Semester) + ' ' + CONVERT(CHAR(4), Year)

  These calculations combine the month, quarter, or semester with the year.

- In the Sales fact table, the named calculation OrderLineDesc combines the order number and the order line using the expression

  CONVERT(CHAR(5),OrderNo) + ' - ' + CONVERT(CHAR(1),OrderLineNo)

5.9.3 Dimensions

Analysis Services supports several types of dimensions as follows:

- A **regular dimension** has a direct one-to-many link between a fact table and a dimension table. An example is the dimension Product.
- A **reference dimension** is indirectly related to the fact table through another dimension. An example is the Geography dimension, which is related to the Sales fact table through the Customer and Supplier dimensions. In this case, Geography may be defined as a reference dimension for the Sales fact table. Reference dimensions can be chained together, for instance, one can define another reference dimension from the Geography dimension.
- In a **role-playing dimension**, a single fact table is related to a dimension table more than once, as studied in Chap. 4. Examples are the dimensions OrderDate, DueDate, and ShippedDate, which all refer to the Date dimension. A role-playing dimension is stored once and used multiple times.
- A **fact dimension**, also referred to as **degenerate dimension**, is similar to a regular dimension but the dimension data are stored in the fact table. An example is the dimension Order.
- In a **many-to-many dimension**, a fact is related to multiple dimension members and a member is related to multiple facts. In the Northwind data warehouse, there is a many-to-many relationship between Employees and Cities, which is represented in the bridge table Territories. This table must be defined as a fact table in Analysis Services, as we will see later.

Dimensions can be defined either from a DSV, which provides data for the dimension, or from preexisting templates provided by Analysis Services. A typical example of the latter is the time dimension, which does not need to be defined from a data source. Dimensions can be built from one or more tables.

In order to define dimensions, we need to discuss how hierarchies are handled in Analysis Services. In the next section, we provide a more detailed discussion on this topic. In Analysis Services, there are two types of hierarchies. **Attribute hierarchies** correspond to a single column in a dimension table, for instance, attribute ProductName in dimension Product. On the other hand, **multilevel** (or **user-defined**) **hierarchies** are derived from two or more attributes, each attribute being a level in the hierarchy, for instance

Product and Category. An attribute can participate in more than one multilevel hierarchy, for instance, a hierarchy Product and Brand in addition to the previous one. Analysis Services supports three types of multilevel hierarchies, depending on how the members of the hierarchy are related to each other: balanced, ragged, and parent-child hierarchies. We will explain how to define these hierarchies in Analysis Services later in this section.

We illustrate next how to define the different kinds of dimensions supported by Analysis Services using the Northwind cube. We start with a **regular dimension**, namely, the Product dimension, shown in Fig. 5.22. The right pane defines the tables in the DSV from which the dimension is created. The attributes of the dimension are given in the left pane. Finally, the hierarchy Categories, composed of the Category and Product levels, is shown in the central pane. The attributes CategoryKey and ProductKey are used for defining these levels. However, in order to show friendly names when browsing the hierarchy, the NameColumn property of these attributes are set to CategoryName and ProductName, respectively.

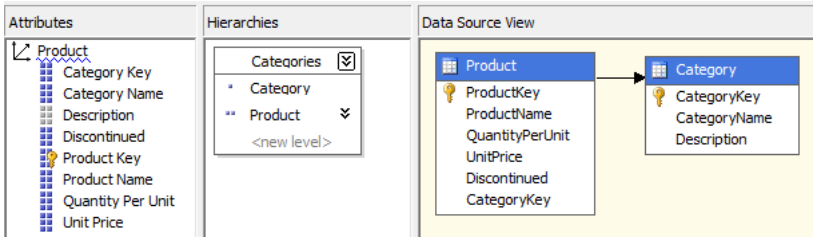


Fig. 5.22. Definition of the Product dimension

Figure 5.23 shows some members of the Product dimension. As shown in the figure, the names of products and categories are displayed in the dimension browser. Notice that a member called Unknown is shown at the bottom of the figure. In fact, every dimension has an Unknown member. If a key error is encountered while processing a fact table, which means that a corresponding key cannot be found in the dimension, the fact value can be assigned to the Unknown member for that dimension. The Unknown member can be made visible or hidden using the dimension property UnknownMember. When set to be visible, the member is included in the results of the MDX queries.

We next explain how the Date dimension is defined in Analysis Services. As shown in Fig. 5.24, the dimension has the hierarchy denoted Calendar, which is defined using the attributes Year, Semester, Quarter, MonthNumber, and DateKey. Since specific MDX functions can be used with time dimensions, the Type property of the dimension must be set to Time. Further, Analysis Services needs to identify which attributes in a time dimension correspond to the typical subdivision of time. This is done by defining the Type property of the attributes of the dimension. Thus, the attributes DayNbMonth, Month-

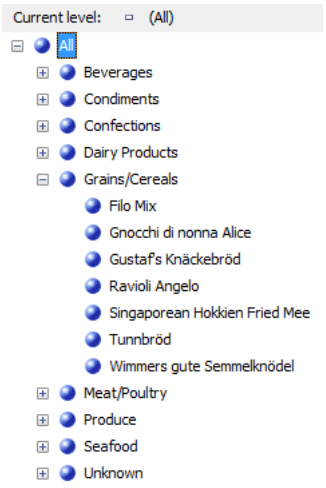


Fig. 5.23. Browsing the hierarchy of the Product dimension

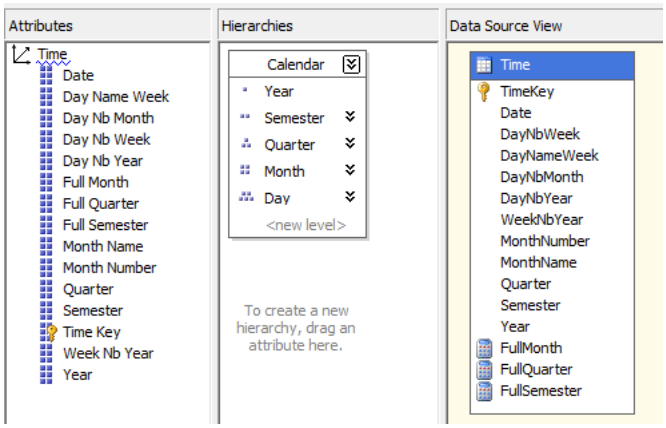


Fig. 5.24. Definition of the Date dimension

Number, Quarter, Semester, and Year are, respectively, of type DayOfMonth, MonthOfYear, QuarterOfYear, HalfYearOfYear, and Year.

Attributes in hierarchies must have a one-to-many relationship to their parents in order to ensure correct roll-up operations. For example, a quarter must roll-up to its semester. In Analysis Services, this is stated by defining a key for each attribute composing a hierarchy. By default, this key is set to the attribute itself, which implies that, for example, years are unique. Nevertheless, in the Northwind data warehouse, attribute MonthNumber has values such as 1 and 2, and thus, a given value appears in several quarters. Therefore, it is necessary to specify that the key of the attribute is a combi-

nation of MonthNumber and Year. This is done by defining the KeyColumns property of the attribute, as shown in Fig. 5.25. Further, in this case, the NameColumn property must also be set to the attribute that is shown when browsing the hierarchy, that is, FullMonth. This should be done similarly for attributes Quarter and Semester.

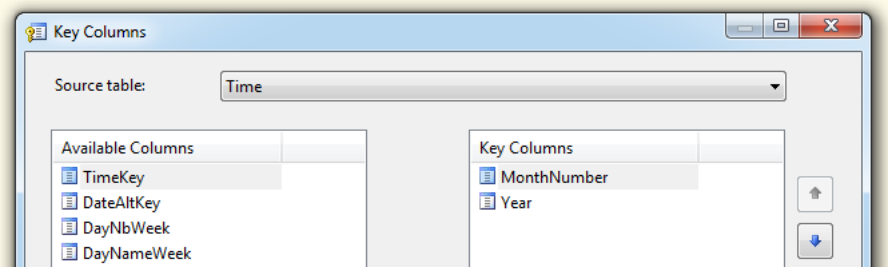


Fig. 5.25. Definition of the key for attribute MonthNumber in the Calendar hierarchy

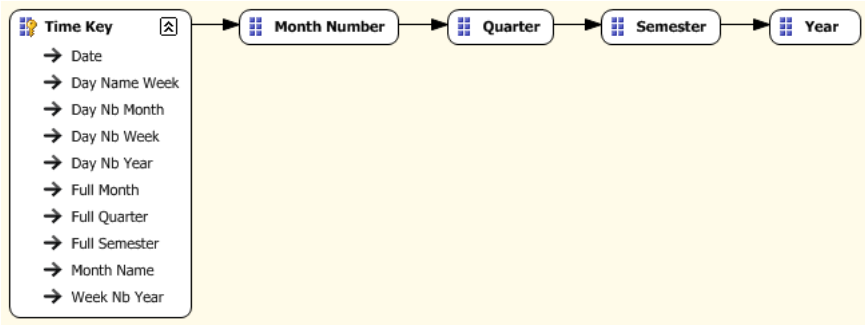


Fig. 5.26. Definition of the relationships in the Calendar hierarchy

When creating a user-defined hierarchy, it is necessary to establish the relationships between the attributes composing such hierarchy. These relationships correspond to functional dependencies. The relationships for the Date dimension are given in Fig. 5.26. In Analysis Services, there are two types of relationships, flexible and rigid. Flexible relationships can evolve over time (e.g., a product can be assigned to a new category), while rigid ones cannot (e.g., a month is always related to its year). The relationships shown in Fig. 5.26 are rigid, as indicated by the solid arrow head.

Figure 5.27 shows some members of the Calendar hierarchy. As can be seen, the named calculations FullSemester, FullQuarter, and FullMonth are displayed when browsing the hierarchy.

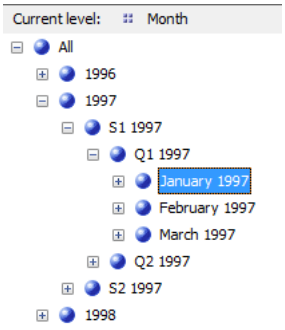


Fig. 5.27. Browsing the hierarchy in the Date dimension

The definition of the **fact dimension** Order follows similar steps than for the other dimensions, except that the source table for the dimension is the fact table. The key of the dimension will be composed of the combination of the order number and the line number. Therefore, the named calculation OrderLineDesc will be used in the NameColumn property when browsing the dimension. Also, we must indicate Analysis Services that this is a degenerate dimension when defining the cube. We will explain this in Sect. 5.9.5.

Finally, in **many-to-many dimensions**, like in the case of City and Employee, we also need to indicate that the bridge table Territories is actually defined as a fact table, so Analysis Services can take care of the double-counting problem. This is also done when defining the cube.

5.9.4 Hierarchies

What we have generically called hierarchies in Chap. 4 and in the present one in Analysis Services are denoted as user-defined or multilevel hierarchies. Multilevel hierarchies are defined by means of dimension attributes, and these attributes may be stored in a single table or in several tables of a snowflake schema. Therefore, both the star and the snowflake and schema representation are supported in Analysis Services.

Balanced hierarchies are supported by Analysis Services. Examples of these hierarchies in the Northwind cube are the Date and the Product dimensions studied above.

Analysis Services does not support **unbalanced hierarchies**. We have seen in Sect. 5.5.2 several solutions to cope with them. On the other hand, Analysis Services supports **parent-child hierarchies**, which are a special case of unbalanced hierarchies. We have seen that such hierarchies define a hierarchical relationship between the members of a dimension. An example is the Supervision hierarchy in the Employee dimension. As can be seen in

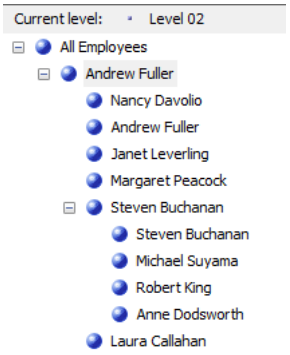


Fig. 5.28. Browsing the Supervision hierarchy in the Employee dimension

Fig. 5.21, in the underlying dimension table the column SupervisorKey is a foreign key referencing EmployeeKey. When defining the dimension, the Usage property for the attributes of the dimension determines how they will be used. In our case, the value of such property will be Parent for the SupervisorKey attribute, will be Regular for all other attributes except EmployeeKey, and will be Key for the attribute EmployeeKey. Figure 5.28 shows the members of the Supervision hierarchy, where the named calculation FullName is displayed when browsing the hierarchy.

In parent-child hierarchies, the hierarchical structure between members is taken into account when measures are aggregated. Thus, for example, the total sales amount of an employee would be her personal sales amount plus the total sales amount of all employees under her in the organization. Each member of a parent-child hierarchy has a system-generated child member that contains the measure values directly associated to it, independently of its descendants. These are referred to as **data members**. The MembersWithData property of the parent attribute controls the visibility of data members: they are shown when the property is set to NonLeafDataVisible, while they are hidden when it is set to NonLeafDataHidden. We can seen in Fig. 5.28 that the data members are visible since both Andrew Fuller and Steven Buchanan appear twice in the hierarchy. The MembersWithDataCaption property of the parent attribute can be used to define a naming template for generating names of data members.

Analysis Services does not support **generalized hierarchies**. If the members differ in attributes and in hierarchy structure, the common solution is to define one hierarchy for the common levels and another hierarchy for each of the exclusive paths containing the specific levels. This is the case for most of the OLAP tools in the market. On the other hand, Analysis Services supports the particular case of **ragged hierarchies**. As we have already seen, in a ragged hierarchy, the parent of a member may be in a level which is not immediately above it. In a table corresponding to a ragged hierarchy,

the missing members can be represented in various ways: with null values or empty strings, or they can contain the same value as their parent.

In Analysis Services, a ragged hierarchy is defined using all of its levels that is, the longest path. To support the display of ragged hierarchies, the HideMemberIf property of a level allows missing members to be hidden. The possible values for this property and their associated behaviors are as follows:

- Never: Level members are never hidden.
- OnlyChildWithNoName: A level member is hidden when it is the only child of its parent and its name is null or an empty string.
- OnlyChildWithParentName: A level member is hidden when it is the only child of its parent and its name is the same as the name of its parent.
- NoName: A level member is hidden when its name is empty.
- ParentName: A level member is hidden when its name is identical to that of its parent.

In order to display ragged hierarchies correctly, the MDX Compatibility property in the connection string from a client application must be set to 2. If it is set to 1, a placeholder member is exposed in a ragged hierarchy.

With respect to **alternative hierarchies**, in Analysis Services several hierarchies can be defined on a dimension, and they can share levels. For example, the alternative hierarchy in Fig. 4.7 will be represented by two distinct hierarchies: the first one composed of Date → Month → CalendarQuarter → CalendarYear and another one composed of Date → Month → FiscalQuarter → FiscalYear.

Analysis Services supports **parallel hierarchies**, whether dependent or independent. Levels can be shared among the various component hierarchies.

Finally, to represent **nonstrict hierarchies** in Analysis Services, it is necessary to represent the corresponding bridge table as a fact table, as it was explained in Sect. 5.9.3. In the relational representation of the Northwind cube given in Fig. 5.4, there is a many-to-many relationship between employees and cities represented by the table Territories. Such a table must be defined as a fact table in the corresponding cube. In this case, using the terminology of Analysis Services, the City dimension has a many-to-many relationship with the Sales measure group, through the Employee intermediate dimension and the Territories measure group.

5.9.5 Cubes

In Analysis Services, a cube is built from one or several DSVs. A cube consists of one or more dimensions from dimension tables and one or more **measure groups** from fact tables. A measure group is composed by a set of **measures**. The facts in a fact table are mapped as measures in a cube. Analysis Services

allows multiple fact tables in a single cube. In this case, the cube typically contains multiple measure groups, one from each fact table.

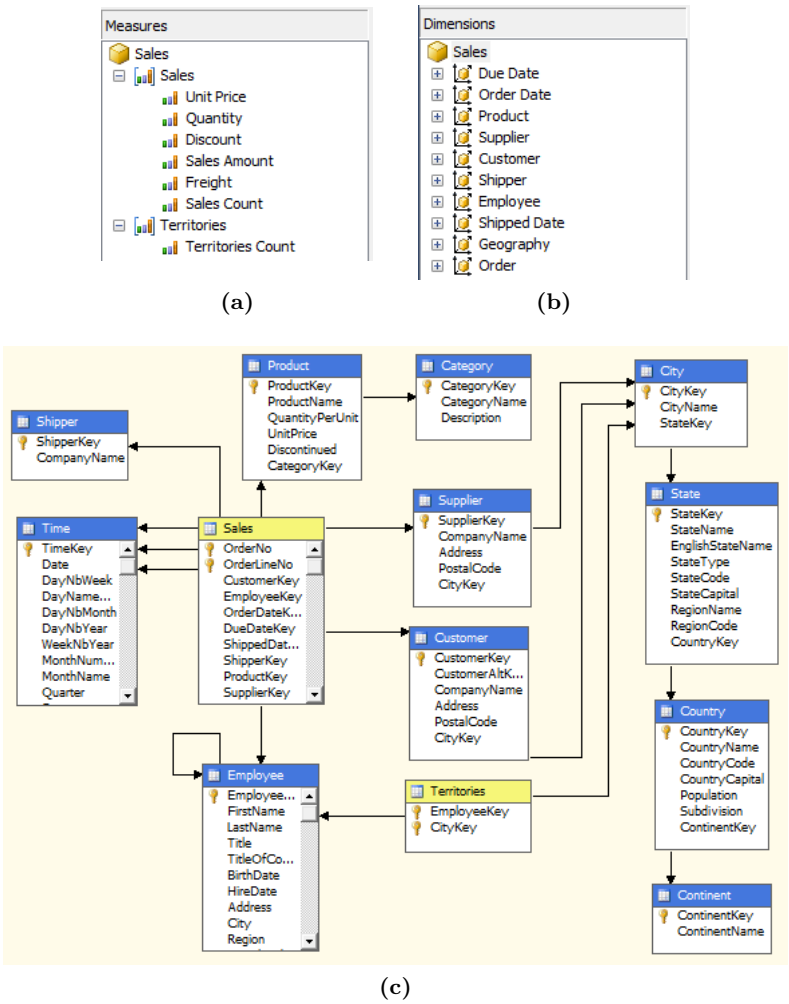


Fig. 5.29. Definition of the Northwind cube in Analysis Services. (**a**) Measure groups; (**b**) Dimensions; (**c**) Schema of the cube

Figure 5.29 shows the definition of the Northwind cube in Analysis Services. As can be seen in Fig. 5.29a, Analysis Services adds a new measure to each measure group, in our case Sales Count and Territories Count, which counts the number fact members associated to each member of each dimension. Thus, Sales Count would count the number of sales for each customer,

supplier, product, and so on. Similarly, Territories Count would count the number of cities associated to each employee.

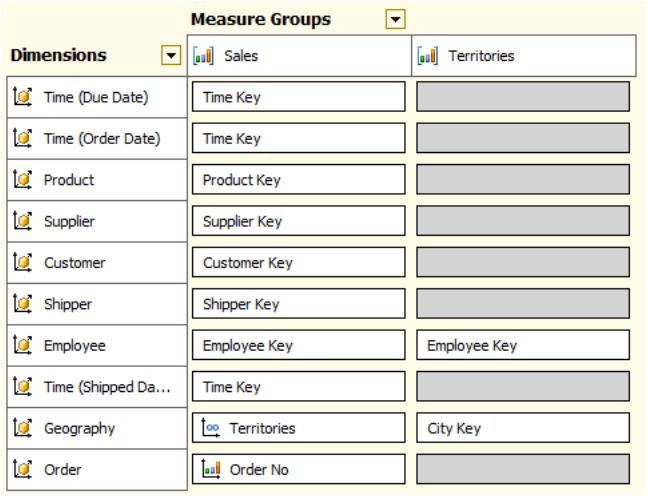


Fig. 5.30. Definition of the dimensions of the Northwind cube

Figure 5.30 shows the relationships between dimensions and measure groups in the cube. With respect to the Sales measure group, all dimensions except the last two are regular dimensions, they do not have an icon to the left of the attribute relating the dimension and the measure group. On the other hand, Geography is a many-to-many dimension linked to the measure group through the Territories fact table. Finally, Order is a fact dimension linked to the measure group through the Order No attribute.

Analysis Services supports the usual additive aggregation functions SUM, MIN, MAX, COUNT, and DISTINCT COUNT. It also supports **semiadditive measures**, that is, measures that can be aggregated in some dimensions but not in others. Recall that we defined such measures in Sect. **??**. Analysis Services provides several functions for semiadditive measures, namely, AverageOfChildren, FirstChild, LastChild, FirstNonEmpty, and LastNonEmpty, among other ones.

The aggregation function associated to each measure must be defined with the AggregationFunction property. The default aggregation measure is SUM, and this is suitable for all measures in our example, except for Unit Price and Discount. Since these are semiadditive measures, their aggregation should be AverageOfChildren, which computes, for a member, the average of its children.

The FormatString property is used to state the format in which the measures will be displayed. For example, measures Unit Price, Sales Amount, and Freight are of type money, and thus, their format will be $###,###.00, where a '#' displays a digit or nothing, a '0' displays a digit or a zero, and ','

and '.' are, respectively, thousand and decimal separators. The format string for measures Quantity and Sales Count will be ###,##0. Finally, the format string for measure Discount will be 0.00%, where the percent symbol '%' specifies that the measure is a percentage and includes a multiplication by a factor of 100.

Further, we can define the default measure of the cube, in our case Sales Amount. As we will see in Chap. **??**, if an MDX query does not specify the measure to be displayed, then the default measure will be used.

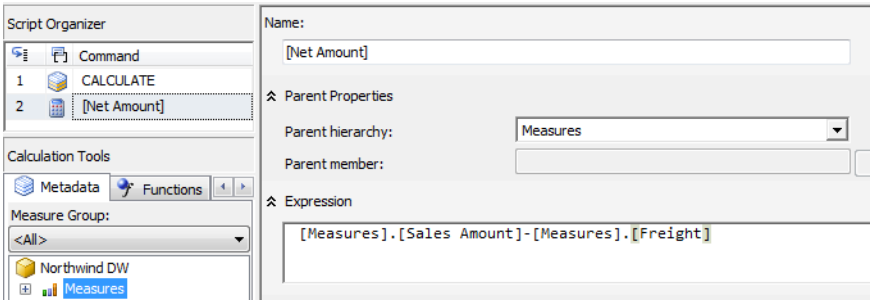


Fig. 5.31. Definition of the Net Amount derived measure

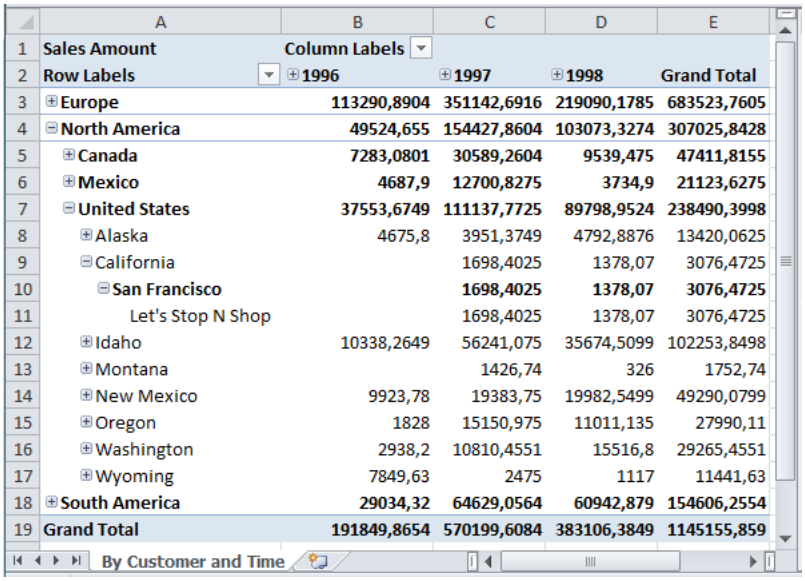
The derived measure Net Amount is defined as shown in Fig. 5.31. As can be seen in the figure, the measure will be a calculated member in the Measures dimension. The defining expression is the difference between the Sales Amount and the Freight measures.

Figure 5.32 shows an example of browsing the Northwind cube in Excel using the PivotTable tools. In the figure, the customer hierarchy is displayed on rows, the time hierarchy is displayed on columns, and the sales amount measure is displayed on cells. Thus, the figure shows the yearly sales of customers at different levels of the geography hierarchy, including the individual sales of a shop located in San Francisco.

5.10 Definition of the Northwind Cube in Mondrian

Mondrian is an open-source relational online analytical processing (ROLAP) server. It is also known as Pentaho Analysis Services and is a component of the Pentaho Business Analytics suite. In this section, we describe Mondrian 4.0, which is the latest version at the time of writing this book.

In Mondrian, a **cube schema** written in an XML syntax defines a mapping between the physical structure of the relational data warehouse and the multidimensional cube. A cube schema contains the declaration of cubes, dimensions, hierarchies, levels, measures, and calculated members. A cube



	A	B	C	D	E
1	Sales Amount	Column Labels			
2	Row Labels	1996	1997	1998	Grand Total
3	Europe	113290,8904	351142,6916	219090,1785	683523,7605
4	North America	49524,655	154427,8604	103073,3274	307025,8428
5	Canada	7283,0801	30589,2604	9539,475	47411,8155
6	Mexico	4687,9	12700,8275	3734,9	21123,6275
7	United States	37553,6749	111137,7725	89798,9524	238490,3998
8	Alaska	4675,8	3951,3749	4792,8876	13420,0625
9	California		1698,4025	1378,07	3076,4725
10	San Francisco		1698,4025	1378,07	3076,4725
11	Let's Stop N Shop		1698,4025	1378,07	3076,4725
12	Idaho	10338,2649	56241,075	35674,5099	102253,8498
13	Montana		1426,74	326	1752,74
14	New Mexico	9923,78	19383,75	19982,5499	49290,0799
15	Oregon	1828	15150,975	11011,135	27990,11
16	Washington	2938,2	10810,4551	15516,8	29265,4551
17	Wyoming	7849,63	2475	1117	11441,63
18	South America	29034,32	64629,0564	60942,879	154606,2554
19	Grand Total	191849,8654	570199,6084	383106,3849	1145155,859

Fig. 5.32. Browsing the Northwind cube in Excel

schema does not define the data source, this is done using a JDBC connection string. We give next the overall structure of a cube schema definition in Mondrian using the Northwind cube.

```
<Schema name="NorthwindDW" metamodelVersion="4.0"
        description="Sales cube of the Northwind company">
    <PhysicalSchema>
        ...
    </PhysicalSchema>
    <Dimension name="Date" table="Date" ... >
        ...
    </Dimension>
    <Cube name="Sales">
        <Dimensions>
            ...
        </Dimensions>
        <MeasureGroups>
            <MeasureGroup name="Sales" table="Sales">
                <Measures>
                    ...
                </Measures>
                <DimensionLinks>
                    ...
                </DimensionLinks>
            </MeasureGroup>
        </MeasureGroups>
    </Cube>
</Schema>
```

The Schema element (starting in line 1) defines all other elements in the schema. The PhysicalSchema element (lines 3–5) defines the tables that are the source data for the dimensions and cubes, and the foreign key links between these tables. The Dimension element (lines 6–8) defines the **shared dimension** Date, which is used several times in the Northwind cube for the role-playing dimensions OrderDate, DueDate, and ShippingDate. Shared dimensions can also be used in several cubes. The Cube element (lines 9–23) defines the Sales cube. A cube schema contains dimensions and measures, the latter organized in measure groups. The Dimensions element (lines 10–12) defines the dimensions of the cube. The measure groups are defined using the element MeasureGroups (lines 13–22). The measure group Sales (lines 14–21) defines the measures using the Measures element (lines 15–17). The DimensionLinks element (lines 18–20) defines how measures relate to dimensions. We detail next each of the elements introduced above.

5.10.1 Schemas and Physical Schemas

The Schema element is the topmost element of a cube schema. It is the container for all its schema elements. A schema has a name and may have other attributes such as description and the version of Mondrian in which it is written. A schema always includes a PhysicalSchema element and one or more Cube elements. Other common elements are Dimension (to define shared dimensions) and Role for access control.

The PhysicalSchema element defines the **physical schema**, which states the tables and columns in the database that provide data for the dimensions and cubes in the data warehouse. The physical schema isolates the logical data warehouse schema from the underlying database. For example, a dimension can be based upon a table that can be a real table or an SQL query in the database. Similarly, a measure can be based upon a column that can be a real column or a column calculated using an SQL expression. The general structure of the physical schema of the Northwind cube is given next.

```
1  <PhysicalSchema>
2      <Table name="Employee" keyColumn="EmployeeKey">
3          <ColumnDefs>
4              <ColumnDef name="EmployeeKey" type="Integer" />
5              <ColumnDef name="FirstName" type="String" />
6              <ColumnDef name="LastName" type="String" />
7              ...
8              <CalculatedColumnDef name="FullName" type="String">
9                  <ExpressionView>
10                     <SQL dialect="generic">
11                         <Column name="FirstName" /> || ' ' ||
12                         <Column name="LastName" />
13                     </SQL>
14                     <SQL dialect="SQL Server">
```

```
15                          <Column name="FirstName" /> + ' ' +
16                          <Column name="LastName" />
17                      </SQL>
18                  </ExpressionView>
19              </CalculatedColumnDef>
20          </ColumnDefs>
21      </Table>
22      ...
23      <Link source="City" target="Employee" foreignKeyColumn="CityKey" />
24      ...
25  </PhysicalSchema>
```

The Table element defines the table Employee (lines 2–21). The columns of the table are defined within the ColumnDefs element, and each column is defined using the ColumnDef element. The definition of the **calculated column** FullName is given in line 8 using the CalculatedColumnDef element. The column will be populated using the values of the columns FirstName and LastName in the underlying database. The ExpressionView element is used to handle the various SQL dialects, which depend on the database system. As can be seen, concatenation of strings is expressed in standard SQL using '||', while it is expressed in SQL Server using '+'. In the case of **snowflake schemas**, the physical schema also declares the foreign key links between the tables using the Link element. In the example above, the link between the tables Employee and City is defined in line 23.

5.10.2 Cubes, Dimensions, Attributes, and Hierarchies

A cube is defined by a Cube element and is a container for a set of dimensions and measure groups, as shown in the schema definition at the beginning of this section (lines 9–23).

A dimension is a collection of attributes and hierarchies. For example, the general structure of the Date dimension in the Northwind cube is given next.

```
1   <Dimension name="Date" table="Date" type="TIME">
2       <Attributes>
3           <Attribute name="Year" keyColumn="Year" levelType="DateYears" />
4           ...
5           <Attribute name="Month" levelType="DateMonths"
6               nameColumn="FullMonth" orderByColumn="MonthNumber" />
7               <Key>
8                   <Column name="Year" />
9                   <Column name="MonthNumber" />
10              </Key>
11          </Attribute>
12          ...
13      </Attributes>
14      <Hierarchies>
15          <Hierarchy name="Calendar" hasAll="true">
```

```
16              <Level attribute="Year" />
17              ...
18              <Level attribute="Month" />
19              ...
20          </Hierarchy>
21      </Hierarchies>
22  </Dimension>
```

The Date dimension is defined in line 1, where attribute type states that this is a time dimension. This would not be the case for the other dimensions such as Customer. In lines 2–13, we define the attributes of the dimension, while the **multilevel hierarchies** are defined in lines 14–21. An Attribute element describes a data value, and it corresponds to a column in the relational model. In line 3, we define the attribute Year based on the column Year, while in lines 5–11 we define the attribute Month based on the column MonthNumber. The nameColumn attribute specifies the column that holds the name of members of the attribute, in this case the calculated column FullMonth, which has values such as September 1997. Further, the orderByColumn attribute specifies the column that specifies the sort order, in this case the column MonthNumber.

The Calendar hierarchy is defined in lines 15–20 using the element Hierarchy. The hasAll="true" statement indicates that the All level is included in the hierarchy. As the attribute of each level in a hierarchy must have a one-to-many relationship with the attribute of the next level, a key must be defined for attributes. For example, since attribute MonthNumber can take the same value for different years, the key of the attribute is defined as a combination of attributes MonthNumber and Year in lines 7–10. This guarantees that all values of the attribute Month are distinct. The key of attribute Year is specified as the column Year using the attribute keyColumn in line 3.

As we will see in Chap. **??**, MDX has several operators that specifically operate over the time dimension. To support these operators, we need to tell Mondrian which attributes define the subdivision of the time periods to which the level corresponds. We indicate this with the attribute levelType in the Attribute element. Values for this attribute can be TimeYears, TimeHalfYears, TimeQuarters, TimeMonths, and so on.

We give next examples of attribute definition of the Product dimension.

```
1  <Attribute name="Unit Price" caption="Prix Unitaire"
2      description="Le prix unitaire de ce produit" keyColumn="UnitPrice" />
3  <Attribute name="Product Name" caption="Nom du Produit"
4      description="Le nom de ce produit" keyColumn="ProductName" />
```

The example shows three properties of the Attribute element. A caption is to be displayed on the screen to a user, whereas the name is intended to be used in code, particularly in an MDX statement. Usually, the name and caption are the same, although the caption can be localized (shown in the language of the user, as in the example) while the name is the same in all languages. Finally, a description is displayed in many user interfaces (such

as Pentaho Analyzer) as tooltips when the mouse is moved over an element. Name, caption, and description are not unique to attributes; the other elements that may appear on user's screen also have them, including Schema, Cube, Measure, and Dimension.

Mondrian implicitly creates **attribute hierarchies**, even if a hierarchy is not defined explicitly for the attribute. For example, if in dimension Employee an attribute is defined as follows:

```
<Attributes>
    <Attribute name="Last Name" keyColumn="LastName" />
    ...
</Attributes>
```

this is interpreted as if the following hierarchy have been defined in the dimension:

```
<Hierarchy name="Last Name">
    <Level attribute="Last Name" />
</Hierarchy>
```

When a schema has more than one cube, these cubes may have several dimensions in common. If these dimensions have the same definitions, they are declared once and can be used in as many cubes as needed. In Mondrian, these are called **shared dimensions**. Further, we have seen that dimensions can be used more than once in the same cube. In the Northwind cube, the Date dimension is used three times to represent the order, due, and shipped dates of orders. We have seen that these are called **role-playing dimensions**. Role-playing dimensions are defined in Mondrian using the concept of shared dimensions as we show below, where the shared dimension is Date.

```
<Cube name="Sales">
    <Dimensions>
        <Dimension name="Order Date" source="Date" />
        <Dimension name="Due Date" source="Date" />
        <Dimension name="Shipped Date" source="Date" />
        <Dimension name="Employee" table="Employee" key="Employee Key">
            ...
        </Dimension>
    </Dimensions>
    ...
</Cube>
```

As we have seen, snowflake dimensions involve more than one table. For example, the Product dimension involves tables Product and Category. When defining dimensions based on **snowflake schemas** in Mondrian, it is necessary to define in which table we can find the dimension attributes, as shown next.

```
<Dimension name="Product" table="Product" key="Product Key">
    <Attributes>
        <Attribute name="Category Name" keyColumn="CategoryName"
```

```
4               table="Category" />
5           ...
6           <Attribute name="Product Name" keyColumn="ProductName" />
7           ...
8       </Attributes>
9       <Hierarchies>
10          <Hierarchy name="Categories" hasAll="true">
11              <Level name="Category" attribute="Category Name" />
12              <Level name="Product" attribute="Product Name" />
13          </Hierarchy>
14      </Hierarchies>
15  </Dimension>
```

As can be seen above, the definition of attribute Category Name states that it comes from table Category (lines 3–4). On the other hand, when defining the attribute Product Name the table is not specified, by default it will be found in the table defined in the Dimension element, that is, the Product table.

We show next how a **parent-child** (or **recursive**) **hierarchy** can be defined in Mondrian using the Supervision hierarchy in the Employee dimension.

```
1   <Dimension name="Employee" table="Employee" key="Employee Key">
2       <Attributes>
3           <Attribute name="Employee Key" keyColumn="EmployeeKey" />
4           ...
5           <Attribute name="Supervisor Key" keyColumn="SupervisorKey" />
6       </Attributes>
7       <Hierarchies>
8           <Hierarchy name="Supervision" hasAll="true">
9               <Level name="Employee" attribute="Employee Key"
10                  parentAttribute="Supervisor Key" nullParentValue="NULL" />
11          </Hierarchy>
12      </Hierarchies>
13  </Dimension>
```

The parentAttribute attribute in line 10 states the name of the attribute that references the parent member in a parent-child hierarchy. The nullParentValue attribute indicates the value determining the top member of the hierarchy, in this case a null value. As in Analysis Services, each member of a parent-child hierarchy has a shadow member, called its **data member**, that keeps the measure values directly associated to it.

As we studied in this chapter and in the previous one, **ragged hierarchies** allow some levels in the hierarchy to be skipped when traversing it. The Geography hierarchy in the Northwind data warehouse shows an example. It allows one to handle, for example, the case of Israel, which does not have states or regions, and the cities belong directly to the country. As in Analysis Services, Mondrian creates hidden members, for example, a dummy state and a dummy region for Israel, to which any city in Israel belongs. Thus, if we ask for the parent member of Tel Aviv, Mondrian will return Israel, the dummy members will be hidden. In short, in Mondrian, when we define a ragged hierarchy, we must tell which members must be hidden. This is done with the hideMemberIf attribute, as shown next.

```
<Dimension name="Customer" table="Customer" />
    <Attributes>
        <Attribute name="Continent" table="Continent"
            keyColumn="ContinentKey" />
        <Attribute name="Country" table="State" keyColumn="CountryKey" />
        <Attribute name="Region" table="State" keyColumn="RegionName" />
        <Attribute name="State" table="State" keyColumn="StateKey" />
        <Attribute name="City" table="City" keyColumn="CityKey" />
        <Attribute name="Customer" keyColumn="CustomerKey" />
        ...
    </Attributes>
    <Hierarchies>
        <Hierarchy name="Geography" />
            <Level attribute name="Continent" />
            <Level attribute name="Country" />
            <Level attribute name="Region" hideMemberIf="IfBlankName" />
            <Level attribute name="State" hideMemberIf="IfBlankName" />
            <Level attribute name="City" />
            <Level attribute name="Customer" />
        </Hierarchy>
    </Hierarchies>
</Dimension>
```

In the schema above, hideMemberIf="IfBlankName" tells that a member in this level does not appear if its name is null, empty, or a whitespace. Other values for the hideMemberIf attribute are Never (the member always appears, the default value) and IfParentName (the member appears unless its name matches the one of its parent).

We next show how a **fact** (or **degenerate**) **dimension** can be defined in Mondrian. Such a dimension has no associated dimension table, and thus, all the columns in the dimension are in the fact table. In the case of the Northwind cube, there is a fact dimension Order, composed by the order number and the order line number corresponding to the fact. To represent this dimension, we may write:

```
<Dimension name="Order" table="Sales">
    <Attributes>
        <Attribute name="Order No" keyColumn="OrderNo">
        <Attribute name="Order Line" keyColumn="OrderLine">
    </Attributes>
</Dimension>
```

Note that the table associated to the dimension is the Sales fact table.

5.10.3 Measures

As in Analysis Services, in Mondrian the measures are also considered dimensions: every cube has an implicit Measures dimension (we will see this in detail in Chap. **??**). The Measures dimension has a single hierarchy, also

called Measures, which has a single level, in turn also called Measures. The measures of the Sales cube are defined as follows:

```
1  <Cube name="Sales">
2      <Dimensions ... />
3      <MeasureGroups>
4          <MeasureGroup name="Sales" table="Sales">
5              <Measures>
6                  <Measure name="Unit Price" column="UnitPrice"
7                      aggregator="avg" formatString="$#,##0.00" />
8                  <Measure name="Sales Count" aggregator="count" />
9                  ...
10             </Measures>
11             <DimensionLinks>
12                 <ForeignKeyLink dimension="Customer"
13                     foreignKeyColumn="CustomerKey" />
14                 <ForeignKeyLink dimension="OrderDate"
15                     foreignKeyColumn="OrderDateKey" />
16                 ...
17                 <FactLink dimension="Order" />
18             </DimensionLinks>
19         </MeasureGroup>
20     </MeasureGroups>
21     <CalculatedMember name="Net Amount" dimension="Measures"
22             formula="[Measures].[Sales Amount]-[Measures].[Freight]">
23         <CalculatedMemberProperty name="FORMAT_STRING"
24             value="$#,##0.00" />
25     </CalculatedMember>
26 </Cube>
```

As can be seen, a **measure** is defined within a **measure group** using a Measure element (lines 5–6). Each measure has a name and an aggregator, describing how to roll up values. The available aggregators are those provided in SQL, that is, SUM, MIN, MAX, AVG, COUNT, and DISTINCT COUNT. A column attribute defines the values to be aggregated. This is required for all aggregators except COUNT. A COUNT aggregator without a column, as for the Sales Count measure in line 9, counts rows.

Mondrian supports **calculated measures**, which are calculated from other measures using an MDX formula. An example is shown for the measure Net Amount (lines 20–24). The dimensions are linked to the measures through the ForeignKeyLink element or, in the case of a fact dimension, through the FactLink element (lines 10–17).

Figure 5.33 shows an example of browsing the Northwind cube in Saiku, an open-source analytics client. In the figure, the countries of customers and the categories of products are displayed on rows, the years are displayed on columns, and the sales amount measure is displayed on cells. Saiku also allows the user to write MDX queries directly on an editor.

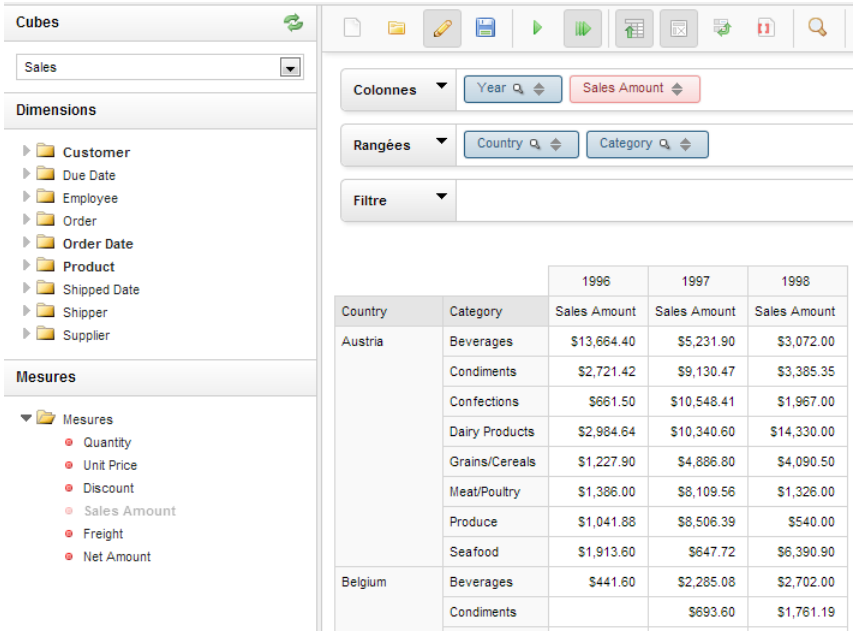


Fig. 5.33. Browsing the Northwind cube in Saiku

5.11 Summary

In this chapter, we have studied the problem of logical design of data warehouses, specifically relational data warehouse design. Several alternatives were discussed: the star, snowflake, starflake, and constellation schemas. Like in the case of operational databases, we provided rules for translating conceptual multidimensional schemas to logical schemas. Particular importance was given to the representation of the various kinds of hierarchies that can occur in practice. The problem of slowly changing dimensions was also addressed in detail. We then explained how the OLAP operations can be implemented on the relational model using the SQL language, and also reviewed the advanced features SQL provides through the SQL/OLAP extension. We concluded the chapter showing how we can implement the Northwind data cube over Microsoft Analysis Services and Mondrian, starting from the Northwind data warehouse.

5.12 Bibliographic Notes

A comprehensive reference to data warehouse modeling can be found in the book by Kimball and Ross [16]. This work covers in particular the topic of slowly changing dimensions. The work by Jagadish et al. [15] discusses the uses of hierarchies in data warehousing. Complex hierarchies like the ones discussed in this chapter were studied, among other works, in [10, 13, 25, 24]. The problem of summarizability is studied in the classic paper of Lenz and Shoshani [19], and in [11, 12]. Following the ideas of Codd for the relational model, there have been attempts to define normal forms for multidimensional databases [18]. Regarding SQL, analytics and OLAP are covered in the books [5, 22]. There is a wide array of books on Analysis Services that describe in detail the functionalities and capabilities of this tool [8, 9, 26, 29, 30]. Finally, a description of Mondrian can be found in the books [3, 4].

References

1. A. Abelló, J. Samos, and F. Saltor. YAM^2 (Yet Another Multidimensional Model): An extension of UML. *Information Systems*, 32(6):541–567, 2006.
2. M. Atkinson, M. Orlowska, P. Valduriez, S. Zdonik, and M. Brodie, editors. *Proceedings of the 25th International Conference on Very Large Data Bases, VLDB'99*, Edinburgh, Scotland, Sept. 1999. Morgan Kaufmann.
3. W. Back, N. Goodman, and J. Hyde. *Mondrian in Action: Open source business analytics*. Manning Publications Co., 2013.
4. R. Bouman and J. van Dongen. *Pentaho Solutions: Business Intelligence and Data Warehousing with Pentaho and MySQL*. Wiley, 2009.
5. J. Celko. *Analytics and OLAP in SQL*. Morgan Kaufmann, 2006.
6. M. Golfarelli and S. Rizzi. A methodological framework for data warehouse design. In I.-Y. Song and T. Teorey, editors, *Proceedings of the 1st ACM International Workshop on Data Warehousing and OLAP, DOLAP'98*, pages 3–9, Bethesda, Maryland, USA, Nov. 1998. ACM Press.
7. M. Golfarelli and S. Rizzi. *Data Warehouse Design: Modern Principles and Methodologies*. McGraw-Hill, 2009.
8. I. Gorbach, A. Berger, and E. Melomed. *Microsoft SQL Server 2008 Analysis Services Unleashed*. Pearson Education, 2009.
9. S. Harinath, R. Pihlgren, D.-Y. Lee, J. Sirmon, and R. Bruckner. *Professional Microsoft SQL Server 2012 Analysis Services with MDX and DAX*. Wrox, 2012.
10. C. Hurtado and C. Gutierrez. Handling structural heterogeneity in OLAP. In R. Wrembel and C. Koncilia, editors, *Data Warehouses and OLAP: Concepts, Architectures and Solutions*, chapter 2, pages 27–57. IRM Press, 2007.
11. C. Hurtado, C. Gutierrez, and A. Mendelzon. Capturing summarizability with integrity constraints in OLAP. *ACM Transactions on Database Systems*, 30(3):854–886, 2005.
12. C. Hurtado and A. Mendelzon. Reasoning about summarizabiliy in heterogeneous multidimensional schemas. In J. Van den Bussche and V. Vianu, editors, *Proceedings of the 8th International Conference on Database Theory, ICDT'01*, number 1973 in Lecture Notes in Computer Science, pages 375–389, London, UK, Jan. 2001. Springer.
13. C. Hurtado and A. Mendelzon. OLAP dimension constraints. In L. Popa, editor, *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, PODS'02*, pages 375–389, Madison, Wisconsin, USA, June 2002. ACM Press.
14. B. Hüsemann, J. Lechtenbörger, and G. Vossen. Conceptual data warehouse design. In M. Jeusfeld, H. Shu, M. Staudt, and G. Vossen, editors, *Proceedings of*

the 2nd International Workshop on Design and Management of Data Warehouses, DMDW'00, page 6, Stockholm, Sweden, June 2000. CEUR Workshop Proceedings.
15. H. Jagadish, L. Lakshmanan, and D. Srivastava. What can hierarchies do for data warehouses. In Atkinson et al. [2], pages 530–541.
16. R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. Wiley, third edition, 2013.
17. J. Lechtenbörger and G. Vossen. Multidimensional normal forms for data warehouse design. *Information Systems*, 28(5):415–434, 2003.
18. W. Lehner, J. Albrecht, and H. Wedekind. Normal forms for multidimensional databases. In M. Rafanelli and M. Jarke, editors, *Proceedings of the 10th International Conference on Scientific and Statistical Database Management, SSDBM'98*, pages 63–72, Capri, Italy, July 1998. IEEE Computer Society Press.
19. H. Lenz and A. Shoshani. Summarizability in OLAP and statistical databases. In Y. Ioannidis and D. Hansen, editors, *Proceedings of the 9th International Conference on Scientific and Statistical Database Management, SSDBM'97*, pages 132–143, Olympia, Washington, USA, Aug. 1997. IEEE Computer Society Press.
20. S. Luján-Mora, J. Trujillo, and I.-Y. Song. A UML profile for multidimensional modeling in data warehouses. *Data & Knowledge Engineering*, 59(3):725–769, 2006.
21. E. Malinowski and E. Zimányi. *Advanced data warehouse design: From conventional to spatial and temporal applications*. Springer, 2008.
22. J. Melton. *Advanced SQL:1999. Understanding Object-Relational and Other Advanced Features*. Morgan Kaufmann, 2003.
23. T. Pedersen. Managing complex multidimensional data. In M.-A. Aufaure and E. Zimányi, editors, *Proceedings of the 2nd European Business Intelligence Summer School, eBISS 2012*, number 138 in Lecture Notes in Business Information Processing, pages 1–28, Brussels, Belgium, July 2012. Springer.
24. T. Pedersen, C. Jensen, and C. Dyreson. Extending practical pre-aggregation in on-line analytical processing. In Atkinson et al. [2], pages 663–674.
25. T. Pedersen, C. Jensen, and C. Dyreson. A foundation for capturing and querying complex multidimensional data. *Information Systems*, 26(5):383–423, 2001.
26. T. Piasevoli. *MDX with Microsoft SQL Server 2008 R2 Analysis Services Cookbook*. Packt Publishing, 2011.
27. E. Pourabbas and M. Rafanelli. Hierarchies. In Rafanelli [28], pages 91–115.
28. M. Rafanelli, editor. *Multidimensional Databases: Problems and Solutions*. Idea Group, 2003.
29. M. Russo, A. Ferrari, and C. Webb. *Expert Cube Development with Microsoft SQL Server 2008 Analysis Services*. Packt Publishing, 2009.
30. M. Russo, A. Ferrari, and C. Webb. *Microsoft SQL Server 2012 Analysis Services: The BISM Tabular Model*. Microsoft Press, 2012.
31. A. Sabaini, E. Zimányi, and C. Combi. Extending the multidimensional model for linking cubes. In E. Zimányi, T. Calders, and S. Vansummeren, editors, *Proceedings of the Journées francophones sur les Entrepôts de Données et l'Analyse en ligne, EDA 2015*, Bruxelles, Belgium, 2015. Editions Hermann.
32. C. Sapia, M. Blaschka, G. Höfling, and B. Dinter. Extending the E/R model for multidimensional paradigm. In T. Ling, S. Ram, and M. Lee, editors, *Proceedings of the 17th International Conference on Conceptual Modeling, ER'98*, number 1507 in Lecture Notes in Computer Science, pages 105–116, Singapore, Nov. 1998. Springer.
33. R. Torlone. Conceptual multidimensional models. In Rafanelli [28], pages 69–90.
34. J. Trujillo, M. Palomar, J. Gomez, and I.-Y. Song. Designing data warehouses with OO conceptual models. *IEEE Computer*, 34(12):66–75, 2001.
35. N. Tryfona, F. Busborg, and J. Borch. StarER: A conceptual model for data warehouse design. In I.-Y. Song and T. Teorey, editors, *Proceedings of the 2nd ACM International Workshop on Data Warehousing and OLAP, DOLAP'99*, pages 3–8, Kansas City, Missouri, USA, Nov. 1999. ACM Press.

36. A. Tsois, N. Karayannidis, and T. Sellis. MAC: Conceptual data modelling for OLAP. In D. Theodoratos, J. Hammer, M. Jeusfeld, and M. Staudt, editors, *Proceedings of the 3rd International Workshop on Design and Management of Data Warehouses, DMDW'01*, page 5, Interlaken, Switzerland, June 2001. CEUR Workshop Proceedings.