

Elzbieta Malinowski · Esteban Zimányi

Advanced Data Warehouse Design

ISBN 978-3-540-74404-7 © 2008 Springer-Verlag Berlin Heidelberg

Errata

Due to an error that occurred during the production process, the figure citations for Fig. 2.3.a-d on pp. 29-32 are incorrect. Figure 3.25 on page 102 and Fig. 7.1 on page 312 are cut.

Furthermore, the captions of Fig. 3.27 on page 104, Fig. 3.31 on page 113 and of Fig. 3.33 on page 115 appear twice. Table 5.2 on page 195 is not printed properly.

The correct pages are attached:

whereas the relational model is a *logical* model targeted toward particular implementation platforms. Several ER concepts do not have a correspondence in the relational model and must be expressed using the only concepts allowed in the model, i.e., relations, attributes, and the related constraints. This translation implies a semantic loss in the sense that data that is invalid in an ER schema is allowed in the corresponding relational schema, unless the relational schema is supplemented by additional constraints. In addition, many such constraints must be hand-coded by the user using mechanisms such as triggers or stored procedures. Furthermore, from the users' perspective, the relational schema is much less readable than the corresponding ER schema. This is crucial when one is considering schemas with hundreds of entity or relationship types and thousands of attributes. This is not a surprise, since this was exactly the reason for devising conceptual models back in the 1970s, i.e., the aim was to better understand the semantics of large relational schemas.

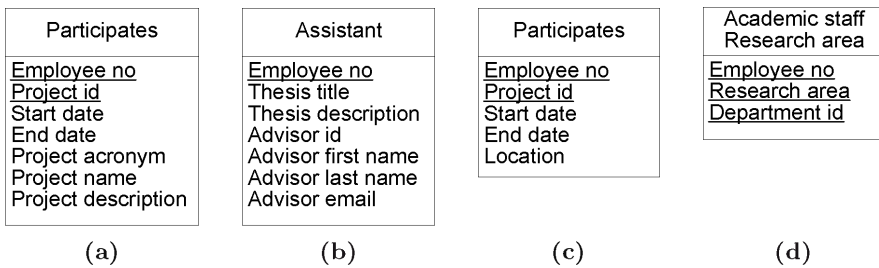


Fig. 2.3. Examples of relations that are not normalized

When one is considering a relational schema, it must be determined whether the relations in the schema have redundancies, and thus may induce anomalies in the presence of insertions, updates, and deletions. Consider for example the relation **Participates** in Fig. 2.3a, which is a variation of the relation with the same name in Fig. 2.2. We can easily verify that the information about a project such as its name, acronym, and description is repeated for each staff member who works on that project. Therefore, when for example the description of a project is to be updated, it must be ensured that all tuples in the relation **Participates** concerning that particular project are given the modified description, otherwise there will be inconsistencies. Similarly, the relation **Assistant** in Fig. 2.3b is also redundant, since the first name, last name, and email address of every professor are repeated for all assistants who have the same advisor. Consider now relation **Participates** in Fig. 2.3c, in which the additional attribute **Location** stores the location of the project. Suppose now that each location is associated with at most one project. In this case, the location information will be repeated for each staff member that works on the project of that location. Finally, consider the relation **Academic**

staff Research area in Fig. 2.3d, where an additional attribute Department id has been added with respect to the relation with the same name in Fig. 2.2. Suppose that members of the academic staff works in several different departments. Since the research areas of staff members are independent of the departments in which they work, there is a redundancy in the above table. Indeed, the information about the research areas of a staff member will be repeated as many times as the number of departments in which he/she works.

Dependencies and normal forms are used to precisely describe the redundancies above. A **functional dependency** is a constraint between two sets of attributes in a relation. Given a relation R and two sets of attributes X and Y in R , a functional dependency $X \rightarrow Y$ holds if and only if, in all the tuples of the relation, each value of X is associated with at most one value of Y . In this case it is said that X *determines* Y . The redundancies in Figs. 2.3a,b,c can be expressed by means of functional dependencies. For example, in the relation Participates in Fig. 2.3a, we have the functional dependency Project id \rightarrow {Project acronym, Project name, Project description}. Also, in the relation Assistant in Fig. 2.3b, the functional dependency Advisor id \rightarrow {Advisor first name, Advisor last name, Advisor email} holds. Finally, in the relation Participates in Fig. 2.3c, there is the functional dependency Location \rightarrow Project id. A key is a particular case of a functional dependency, where the set of attributes composing the key functionally determines all of the attributes in the relation.

The redundancy in the relation Academic staff Research areas in Fig. 2.3d is captured by another kind of dependency. Given two sets of attributes X and Y in a relation R , a **multivalued dependency** $X \twoheadrightarrow Y$ holds if the value of X determines a set of values for Y , independently of any other attributes. In this case it is said that X *multidetermines* Y . In the relation in Fig. 2.3d, we have the multivalued dependencies Employee no \twoheadrightarrow Research area, and consequently Employee no \twoheadrightarrow Department id. It is well known that functional dependencies are special cases of multivalued dependencies, i.e., every functional dependency is also a multivalued dependency. A multivalued dependency $X \twoheadrightarrow Y$ is said to be trivial if either $Y \subseteq X$ or $X \cup Y = R$, otherwise it is nontrivial.

A **normal form** is an integrity constraint certifying that a relational schema satisfies particular properties. Since the beginning of the relational model in the 1970s, many types of normal forms have been defined. In addition, normal forms have also been defined for other models, such as the entity-relationship model and the object-relational model. In the following, we consider only four normal forms that are widely used in relational databases.

As already said, the relational model allows only attributes that are atomic and monovalued. This restriction is called the **first normal form**. As we shall see in Sect. 2.3.2, the object-relational model removes this restriction and allows composite and multivalued attributes.

The **second normal form** avoids redundancies such as those in the table *Participates* in Fig. 2.3a. In order to define the second normal form, we must define the following concepts:

- A **prime attribute** is an attribute that is part of a key.
- A **full functional dependency** is a dependency $X \rightarrow Y$ in which the removal of an attribute from X invalidates the dependency.

Now we can give the definition of the second normal form: A relation schema is in the second normal form if every nonprime attribute is fully functionally dependent on every key. As we can see, the table *Participates* above is not in the second normal form, since *Project acronym*, *Project name*, and *Project description* are nonprime attributes (they do not belong to a key) and are dependent on *Project id*, i.e., on part of the key of the relation. To make the relation comply with the second normal form, the nonprime attributes dependent on *Project id* must be removed from the table and an additional table *Project*(*Project id*, *Project acronym*, *Project name*, *Project description*) must be added to store the information about projects.

The **third normal form** avoids redundancies such as those in the table *Assistant* in Fig. 2.3b. In order to define the third normal form, we must define one additional concept:

- A dependency $X \rightarrow Z$ is **transitive** if there is a set of attributes Y such that the dependencies $X \rightarrow Y$ and $Y \rightarrow Z$ hold.

Now we can give the definition of the third normal form: A relation is in the third normal form if it is in the second normal form and there are no transitive dependencies between a key and a nonprime attribute. The table *Assistant* above is not in the third normal form, since there is a transitive dependency from *Employee no* to *Advisor id*, and from *Advisor id* to *Advisor first name*, *Advisor last name*, and *Advisor email*. To make the relation comply with the third normal form, the attributes dependent on *Advisor id* must be removed from the table and an additional table *Advisor*(*Advisor id*, *Advisor acronym*, *Advisor first name*, *Advisor last name*) must be added to store the information about advisors.

The **Boyce-Codd normal form** avoids redundancies such as those in the table *Participates* in Fig. 2.3c. Recall that in this case it is supposed that there is a functional dependency $\text{Location} \rightarrow \text{Project id}$. A relation is in the Boyce-Codd normal form with respect to a set of functional dependencies F if, for every nontrivial dependency $X \rightarrow Y$ that can be derived from F , X is a key or contains a key of R . The table *Participates* above is not in the Boyce-Codd normal form, since the above functional dependency holds and *Location* is not a key of the relation. To make the relation comply with the Boyce-Codd form, the attribute *Location* must be removed from the table, and an additional table *LocationProject*(*Location*, *Project id*) must be added to store the information about the project associated with each location. Note that all relations in Fig. 2.2 are in the Boyce-Codd normal form.

The **fourth normal form** avoids redundancies such as those in the table Academic staff Research area in Fig. 2.3d. A relation is in the fourth normal form with respect to a set of functional and multivalued dependencies F if, for every nontrivial dependency $X \twoheadrightarrow Y$ that can be derived from F , X is a key or contains a key of R . The table above is not in the fourth normal form, since there are multivalued dependencies from Employee no to Research area, and from Employee no to Department id, and Employee no is not a key of the relation. To make the relation comply with the fourth normal form, the attribute Department id must be removed from the table, and an additional table AcademicStaffDepart(Employee, Department id) must be added to store the information about the departments in which a member of the academic staff works.

2.3.2 The Object-Relational Model

As shown in the previous section, the relational model suffers from several weaknesses that become evident when we deal with complex applications.

- The relational model provides a very simple data structure (i.e., a relation), which disallows multivalued and complex attributes. Therefore, in a relational database, complex objects must be split into several tables. This induces performance problems, since assembly and disassembly operations using joins are needed for retrieving and storing complex objects in a relational database.
- The set of types provided by relational DBMSs is very restrictive. It includes only some basic types such as integer, float, string, and date, and uninterpreted binary streams that must be manipulated explicitly by the user. Such a restricted set of types does not fit complex application domains.
- There is no integration of operations with data structures, i.e., there is no encapsulation, and no methods associated with a table.
- Since there is no possibility to directly reference an object by use of a surrogate or a pointer, every link between tables is based on comparison of values. Therefore, joins represent a bottleneck with respect to performance.

During the 1980s, a considerable amount of research addressed the issue of relaxing the assumption that relations must satisfy the first normal form. Many results for the relational model were generalized to such an extended model, called **non-first-normal-form model**, or NFNF or NF2 model (e.g., [11, 262]). Such research has been introduced into the database standard SQL:2003 [192, 193] under the name of the object-relational model. In addition, current database management systems such as Oracle, Informix, DB2, and PostgreSQL have also introduced object-relational extensions, although these do not necessarily comply with the SQL:2003 standard.

The object-relational model preserves the foundations of the relational model, while extending its modeling power by organizing data using an object

representing the situation where several members of a dimension participate in the same instance of a fact relationship. A common example used to represent this situation is an analysis of clients' balances in bank accounts, as shown in Fig. 3.25.

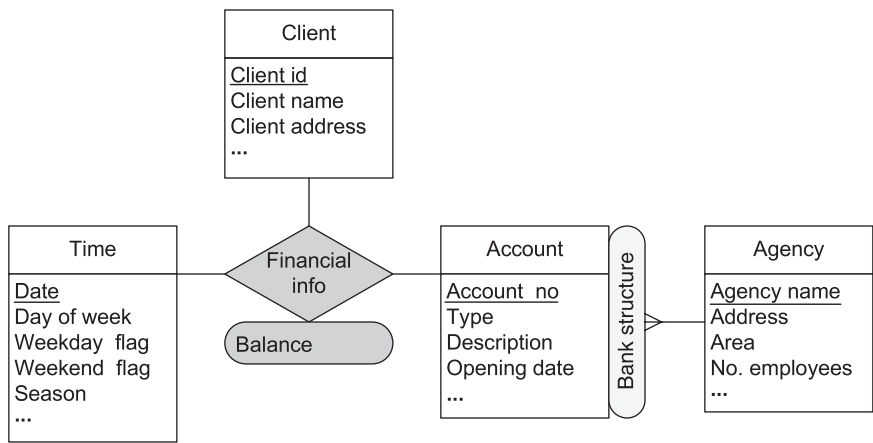
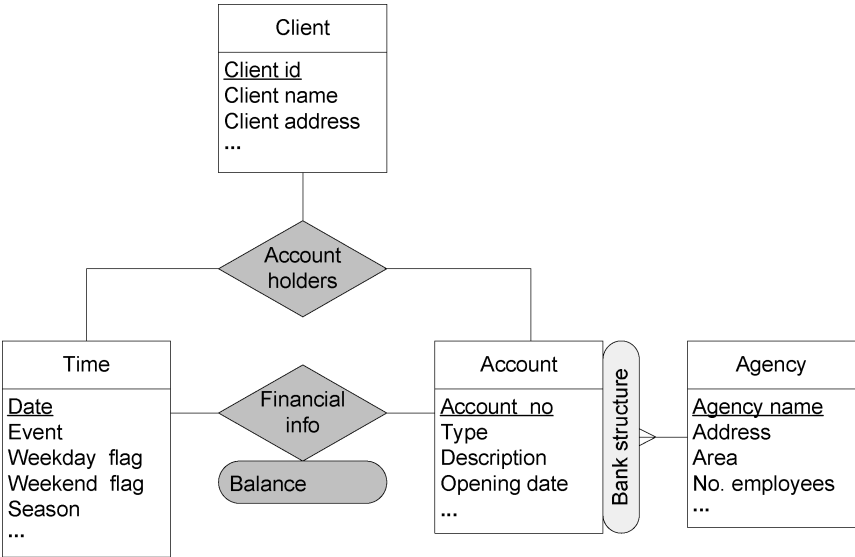


Fig. 3.25. Multidimensional schema for analysis of bank accounts

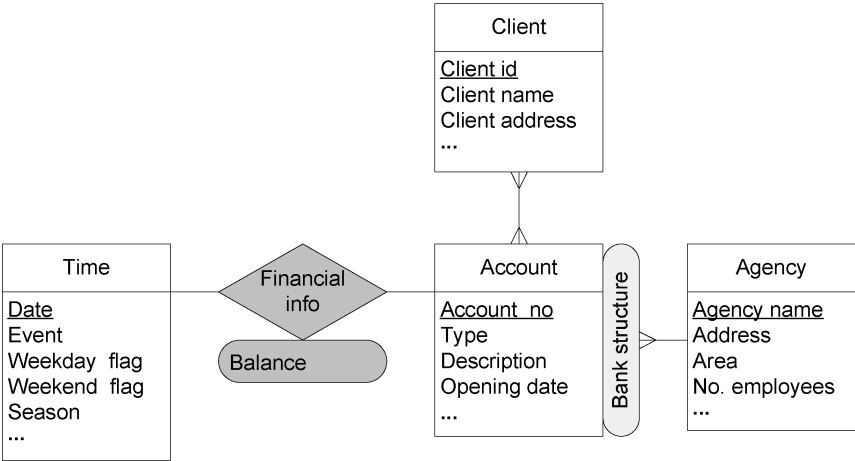
Since an account can be jointly owned by several clients, aggregation of the balance according to the clients will count this balance as many times as the number of account holders. Let us consider the example in Fig. 3.26. At some point in time T1, we have two accounts A1 and A2 with balances of, respectively, 100 and 500. As shown in the figure, both accounts are shared between several clients: account A1 between C1, C2, and C3, and account A2 between C1 and C2. The total balance of the two accounts is equal to 600; however, aggregation (for example, according to the Time or the Client dimension) gives a value equal to 1300.

Time	Account	Client	Balance
T1	A1	C1	100
T1	A1	C2	100
T1	A1	C3	100
T1	A2	C1	500
T1	A2	C2	500

Fig. 3.26. Example of double-counting problem for a multivalued dimension



(a) Creating two fact relationships



(b) Including a nonstrict hierarchy

Fig. 3.27. Decomposition of the fact relationship in Fig. 3.25

since the two fact relationships represent different granularities, queries with drill-across operations are complex, demanding a conversion either from a finer to a coarser granularity (for example, grouping clients to know who holds a specific balance in an account) or vice versa (for example, distributing a balance between different account holders). Note also that the two schemas in Fig. 3.27 could represent the information about the percentage of ownership

- **Normalized tables or snowflake structure:** each level is represented as a separate table that includes a key and the descriptive attributes of the level. For example, using Rules 1 and 2a of Sect. 3.5.2 for the **Product** groups hierarchy in Fig. 3.4 gives a snowflake structure with tables **Product**, **Category**, and **Department**, as shown in Fig. 3.31a.
- **Denormalized or flat tables:** the key and descriptive attributes of all levels forming a hierarchy are included in one table. This structure can be obtained in two ways: (1) denormalizing the tables that represent several hierarchy levels (for example, including in one table all attributes of the **Product**, **Category**, and **Department** tables shown in Fig. 3.31a), or (2) mapping a dimension that includes a one-level hierarchy according to Rule 1 (for example, the **Store** dimension in Fig. 3.2 may be represented as shown in Fig. 3.31b).

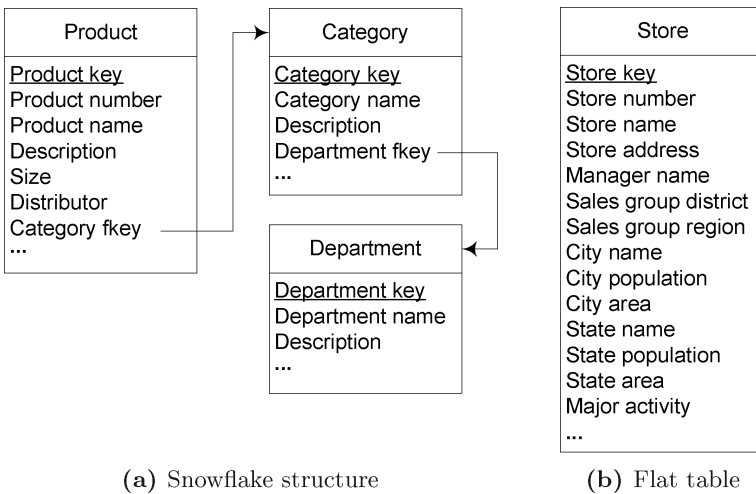


Fig. 3.31. Relations for a balanced hierarchy

Normalized tables are used in **snowflake schemas** (see Sect. 2.7). They represent hierarchical structures better, since every level can be easily distinguished and, further, levels can be reused between different hierarchies. Additionally, this representation can easily manage heterogeneity across levels [129], i.e., it allows different levels of a hierarchy to include specific attributes. For example, the **Product**, **Category**, and **Department** tables in Fig. 3.31a have specific attributes. This data structure allows measures to be aggregated using, for example, the SQL group by, roll-up, or cube operators (see Sect. 2.6.3). Further, in some applications, snowflake schemas can improve system performance in spite of requiring join operations between the relations representing

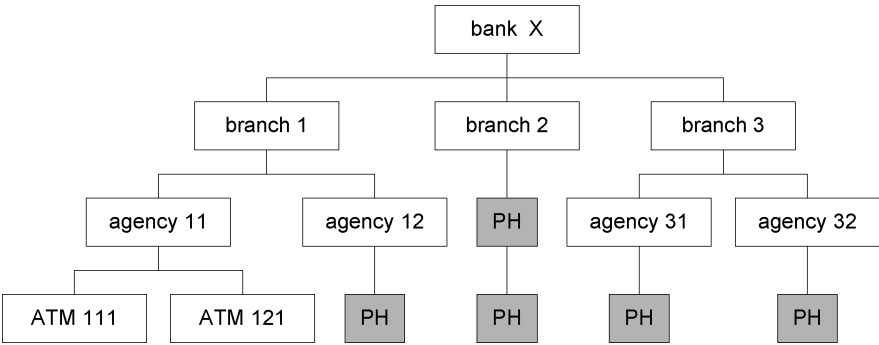


Fig. 3.32. Transformation of the unbalanced hierarchy shown in Fig. 3.5b into a balanced one using placeholders

- 4. The unnecessary introduction of meaningless values requires more storage space.
- 5. A special interface needs to be implemented to hide placeholders from users.

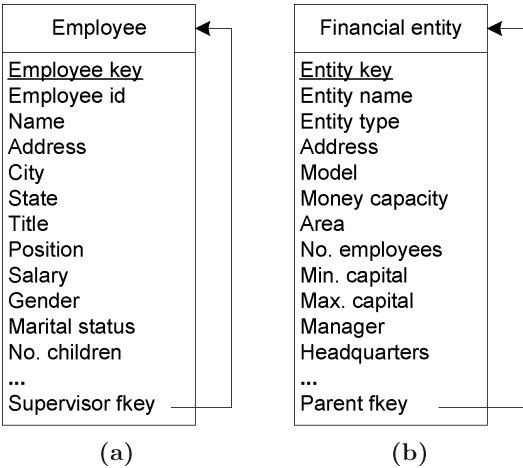


Fig. 3.33. Relational implementation of the recursive hierarchies shown in Fig. 3.6

Recall from Sect. 3.2.1 that **recursive hierarchies** are a special case of unbalanced hierarchies. Mapping recursive hierarchies to the relational model yields **parent-child tables** containing all attributes of a level, and an additional foreign key relating child members to their corresponding parent. Figure 3.33a shows the table representing the recursive hierarchy shown in Fig. 3.6a.

Table 5.2. Temporality types of the MultiDim model

Temporality types	Levels	Attributes	Measures	Parent-child relationships
LS	✓	✗	✗	✓
VT	✗	✓	✓	✗
TT	✓	✓	✓/✗	✓
LT	✓	✓	✓	✓

5.4 Temporal Support for Levels

Changes in a level can occur either for a member as a whole (for example, inserting or deleting a product in the catalog of a company) or for attribute values (for example, changing the size of a product). Representing these changes in a temporal data warehouse is important for analysis purposes, for example to discover how the exclusion of some products or changes to the size of a product influence sales. As shown in Fig. 5.6, in the MultiDim model, a level may have temporal support independently of the fact that it has temporal attributes.

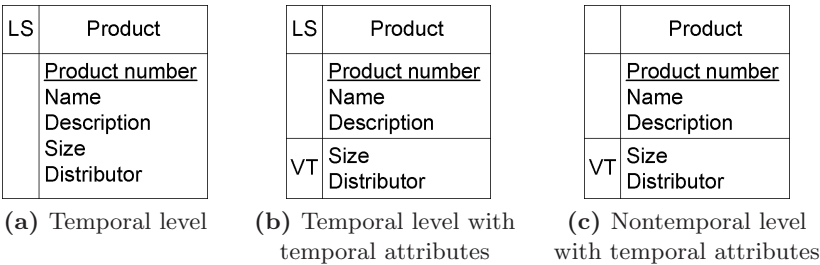


Fig. 5.6. Types of temporal support for a level

Temporal support for a level allows a time frame to be associated with its members. This is represented by including the symbol for the temporality type next to the level name, as shown in Fig. 5.6a. Various temporality types are possible for levels. Lifespan support is used to store the time of existence of the members in the modeled reality. On the other hand, transaction time and loading time indicate when members are current in a source system and in a temporal data warehouse, respectively. These three temporality types can be combined.

On the other hand, temporal support for attributes allows one to store changes in their values and the times when these changes occurred. This is

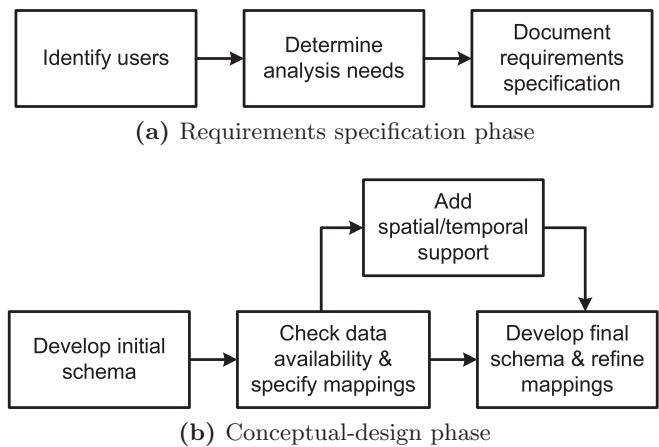


Fig. 7.1. Steps of the analysis-driven approach for spatial and temporal data warehouses

warehouse and to determine the analysis needs, which are collected in the second step. The information gathered and the corresponding metadata are documented in the third step and serve as a basis for the next phase.

The conceptual-design phase (Fig. 7.1b) starts with the development of the initial spatial-data-warehouse schema. Note that this schema already includes spatial elements, since we assume that the users are able to refer to spatial data when expressing their specific analysis needs. Therefore, we follow the lower path of Fig. 7.1b. In the following step, it must be determined whether the data is available in the source systems, and the corresponding mappings with data warehouse elements are established. Note, however, that external sources may be needed if the required spatial support does not exist in the source systems. During the last phase, the final schema is developed; it includes all data warehouse elements, for which the corresponding data exists in the source systems (whether internal or external). Additionally, the corresponding mappings between the two kinds of systems is delivered.

We now illustrate this approach with our risk management application. In order to determine the analysis requirements, a series of interviews was conducted, targeting users at different organizational levels, i.e., from senior risk experts to field surveyors who inspect damage after a hazard event. From these interviews it was established that owing to the increasing number of hazard events, a reclassification of the various risk zones and land use zones had to be performed. The various analysis scenarios that were elicited were as follows:

1. The evolution in time of the extent and frequency of hazard events for the various types of risk zones (red, blue, and white) in different land plots.

Advanced Data Warehouse Design
From Conventional to Spatial and Temporal Applications
Malinowski, E.; Zimányi, E.
2008, XXI, 435 p., Hardcover
ISBN: 978-3-540-74404-7