Using the DBV model to maintain versions of multi-scale geospatial data

João Sávio C. Longo, Luís Theodoro O. Camargo,
Claudia Bauzer Medeiros, André Santanchè

Institute of Computing (IC)
University of Campinas (UNICAMP)
Campinas, SP – Brazil

{joaosavio,theodoro}@lis.ic.unicamp.br
{cmbm,santanche}@ic.unicamp.br

December 18, 2012

Abstract

Work on multi-scale issues concerning geospatial data presents countless challenges that have been long attacked by GIScience researchers. Indeed, a given real world problem must often be studied at distinct scales in order to be solved. Most implementation solutions go either towards generalization (and/or virtualization of distinct scales) or towards linking entities of interest across scales. In this context, the possibility of maintaining the history of changes at each scale is another factor to be considered. This paper presents our solution to these issues, which accommodates all previous research on handling multiple scales into a unifying framework. Our solution builds upon a specific database version model – the multiversion MVDB – which has already been successfully implemented in several geospatial scenarios, being extended here to support multi-scale research. The paper also presents our implementation of of a framework based on the model to handle and keep track of multi-scale data evolution.

multi-scale, database versions, MVDB model

1 Introduction

A major challenge when dealing with geospatial data are the many scales in which such data are represented. For instance, national mapping agencies produce multi-scale[1] geospatial data and one of the main difficulties is to guarantee

[1] Unless specified, this paper uses the term "scale" to refer to cartographic scale.

consistency between the scales [SVvO+11]. Most research efforts either concentrate on modeling or on data structures/database aspects.

Literature on the management of geospatial data at multiple scales concentrates on two directions: (a) generalization algorithms, which are mostly geared towards handling multiple spatial scales via algorithmic processes, that may, for instance, start from predefined scales, or use reactive behaviors (e.g., agents) to dinamically compute geometric properties; and (b) multi-representation databases (MRDBs), which store some predefined scales and link entities of interest across scales, or multiple representations within a scale. These two approaches roughly correspond to Zhou and Jones' [ZJ03] multi-representation spatial databases and linked multiversion databases[2].

While generalization approaches compute multiple virtual scales, approaches based on data structures, in which we will concentrate, rely on managing stored data. From this point of view, options may vary from maintaining separate databases (one for each scale) to using MRDBs, or MRMS (Multiple Representation Management Systems) [FJ03]. MRDBs and MRMS concern data structures to store and link different objects of several representations of the same entity or phenomenon [Sar07]. They have been successfully reported in, for instance, urban planning, or in the aggregation of large amounts of geospatial data and in cases that applications require data in different levels of detail [Oos09, GZH+10, PSVZ09]. Oosterom et al. [OS10], in their multi-representation work, also comment on the possibility of storing the most detailed data and computing other scales via generalization. This presents the advantage of preserving consistency across scales (since all except for a basis are computed). Generalization solutions vary widely, but the emphasis is on real time computation, which becomes costly if there are continuous updates to the data – e.g., see the hierarchical agent approach of [RD07] or the multiple representations of [BBB07].

This paper presents our approach to manage multiple scales of geospatial objects that is based on extending the DBV (*Database Version*) model [CJ90, GJ01] to provide support to flexible MRDB structures. As will be seen, our extension (and its implementation) provide the following advantages to other approaches: (a) it supports keeping track of evolution of objects at each scale, and across scales, simultaneously; (b) it provides management of multi-scale objects saving storage space [CJ90], as opposed to approaches in which evolution requires replication; and (c) it supports evolution according to scale and to shape, where the latter can be treated as alternative versioning scenarios.

2 Basic Conceps and Related Work

2.1 MRDB and Multi-Scale Data

Spaccapietra et al. [SPV00] cite that in different scales the objects are usually represented in different ways, because each scale can have a convention of

[2]We point out that our definition of *version* is not the same as that of Zhou and Jones

representation. Objects can appear/disappear or be aggregated/disaggregated, shapes can be simplified or objects could not appear in some scales.

Relying of this fact, MRDBs (Multiple Representation Database) have been proposed to solve this problem. These are data structures to store and link different objects of several representations of the same entity or phenomenon [Sar07]. There are plenty of benefits to this approach, according to Sarjakoski [Sar07]:

- Maintenance is flexible, since more specific level updates can be propagated to the lower resolution data;
- The links between objects of different levels of representation can provide a basis for consistency and automatic error checking;
- MRDBs can be used for multi-scale analysis of spatial information, such as comparing data at different resolution levels.

According to Deng et al. [DWL08], there are three main variants to link objects in an MRDB. The first one is called *attribute variant* and all data are stored in one dataset. The second variant, named *bottom-up variant*, considers the existence of two or more datasets, linked by an additional attribute that links the objects of the actual scale to those of the immediately smaller scale. The *top-down variant*, the third approach, is similar to the second, except for the fact that the link points to the immediately larger scale.

As an example of implementation, Parent et al. [PSZ06] present MurMur, an effort to develop a manipulation approach to geographic databases that have multiple representations. Additional research on MRDB structures includes Burghardt et al.'s work [BPB10], which shows how to improve the creation of maps via automated generalization for topographic maps and multi-representation databases.

Although MRDB structures are used to treat multi-representation problems, this paper proposes to deal with multi-scale problems, a subset of those related to multi-representations. Our proposal allows keeping the history of changes within and across scales, which is not directly supported by MRDBs.

2.2 The DBV Model

The DBV (*Database Version*) model is an approach to "maintain consistency of object versions in multiversion database systems" [CJ90]. A DBV represents a possible state or version of the database [CJ90]. It can be seen as a virtual view of a database in which multiple versions of objects are stored. This view shows just one version of each object, so that users can work at each DBV as if they were handling a constant (monoversion) state of the database. Temporal versioning is just one type of version. Database researches and, more specifically, the DBV model, consider a version to be any stored modification of a (database) state. Thus, a given real world object mat be versioned in time, but also different simultaneous representations are versions of that object.

In this model, there are two levels: the logical and the physical. The first corresponds to the user view of each database state (DBV) and is represented by the *logical versions*. The second is represented by the *physical versions* of the stored objects.

A *multiversion object* represents one single entity in the real world – any attribute (geometry, color, etc) may change; as long as the experts consider it to be the same entity, it is not assigned a new *id*. Let us consider a *multiversion object* *o*, e.g. a car, with two different models, one painted blue and other red. Internally, the database will store the *physical versions* of *o* as *pv1* and *pv2*. Logically, *pv1* will appear in one DBV and *pv2* in another. The physical database will have cars of both colors, but from a logical (user's) point of view, only one color exists.

Versions are organized in a derivation tree as Figure 1 shows. Each version is associated with a *stamp* value (0, 0.1, etc). The derivation tree indicates how DBVs are derived from each other, thus supporting change traceability. Derivations always correspond to some kind of update. For instance, Figure 1 shows that DBV *d1* (*stamp* 0.1) is derived from *d0* (*stamp* 0) and that *d2* (*stamp* 0.1.1) and *d3* (*stamp* 0.1.2) are derived from *d1*. By definition, there is no data in *stamp* 0 (root).

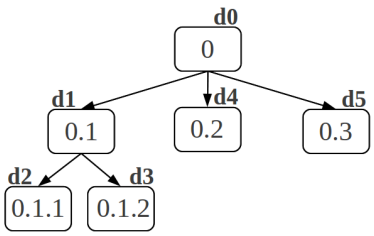


Figure 1: Derivation tree of database versions

One of the main advantages of using the DBV approach is that only the changes must be stored. Data that are not modified are shared from previous DBVs through semantics of the version *stamps*. For instance, suppose we have to access all *logical versions* related to *d2*. It is also necessary to look up at all previous DBVs up to the root – *d1*, since each version stores only the data changes. More information about the DBV model can be seen in [CJ90, GJ01].

3 Our Approach

3.1 Overview

We have adopted the DBV model to support multiple scales. Each DBV represents a particular scale. The set of DBVs, which can be interlinked, correspond to a multi-scale/multi-representation world.

We extended the model so that, instead of one derivation tree, each scale has its own tree and all trees evolve together. Besides the version *stamp*, each DBV

has an associated scale *s*. We use the following notation: the DBV *d0* of scale 1 as $d0_1$. Figure 2 shows four versions (0, 0.1, 0.1.1 and 0.1.2) and *n* scales.

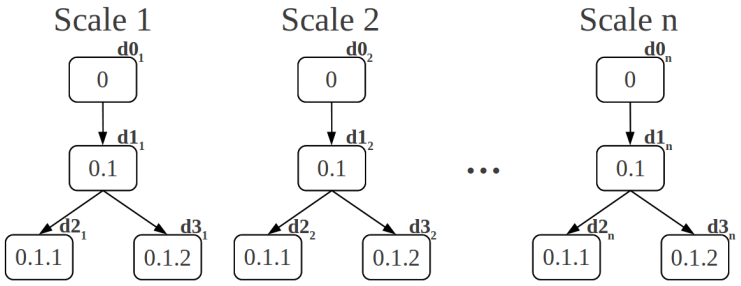


Figure 2: Example of our approach to maintain versions of multi-scale geospatial data

Let a real world object *o1* be physically stored in a database in two scales, receiving physical identifiers *pv1* and *pv2*, where *pv1* is a polygon and *pv2* a point. Polygon and point are respectively represented in DBVs $d1_1$ and $d1_2$. Using this information and the DBV concepts, we have two *logical versions* (each in a DBV) represented in the following way: *logical version 1* = ((*o1*, $d1_1$), *pv1*) and *logical version 2* = ((*o1*, $d1_2$), *pv2*). In other words, DBV $d1_1$ contains the polygon version of *o1*, and $d1_2$ the point version of *o1*.

Unlike several multi-representation approaches, we do not link explicitly objects of different scales (e.g., *pv1* and *pv2*). Instead, the link is achieved implicitly by combining *stamp* and derivation trees, using the concept of *logical versions*. This kind of link is similar to the *bottom-up variant* seen in section 2.1.

A change in a real world that requires creating a new version in scale *s* may require changes in other scales. Keeping one tree per scale, moreover, makes sense because, as remarked by [SPV00], for large scale changes an object suffers radical changes when scale changes occur and thus there is seldom any intersection (if any) between DBVs in different scales. To simplify maintaining consistency across scales, we postulate that all derivation trees grow and shrink together and have the same topology. This leads to the notion of multi-scale *scenario* σ, for short, *scenario*. Each *scenario* is formed by all the DBVs with the same version *stamp*. For instance, in Figure 2, $d0_1$, $d0_2$, ..., $d0_n$ form a *scenario*, and so do $d1_1$, $d1_2$, ..., $d1_n$; etc. In fact, there may be many *scenarios*.

For managing the versions, we use the propagation algorithm adopted by the DBV model: only data changes must be stored and unchanged data are propagated across versions.

3.2 The Model

Figure 3 represents our model in UML. We introduce a new class called *Scale*, which has an identifier named *sid* (*scale id*). A DBV is identified by the couple (*stamp*, *sid*). The *Scale* class allows the association of a DBV with different

types of scales, where spatial scale is one of them (another example is the temporal scale[3]). *LogicalVersion* class associates a *MultiversionObject* to a *DBV*. A *physical version* of an object underlies a *logical version* (i.e., it may appear in some DBV). This is expressed by the relationship between *LogicalVersion* and *PhysicalVersion* classes. The latter is the root of a hierarchy of classes of all kinds of objects that can be versioned (see Figure 5 later on) and allows the user to choose which data will be versioned though the *OBJTYPE* parameterized type. This approach forces the subclasses of *PhysicalVersion* to provide the data to be versioned. If a *multiversion object o* does not appear in DBV *d*, we represent this situation setting the *PhysicalVersion* as *null*. A DBV has one parent and – by a derivation process – one or more children.

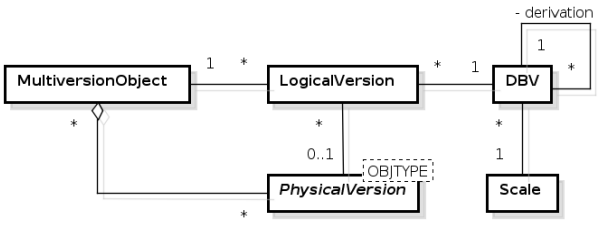


Figure 3: Our basic model in UML

The model considers the following operations: (a) Create, modify and delete a *multiversion object* and its *physical versions*; (b) Create a new *Scale* (which will create a new tree); (c) Create or remove a DBV (affecting the trees); (d) Access a DBV (gathering the relevant objects of interest).

4 Implementation Details

4.1 Overview

We chose to implement our framework in an object-relational database due its widespread adoption and to its support of geospatial features. We developed an API[4] on top of the PostGIS[5] spatial database extension for PostgreSQL[6]. Our implementation uses the Java programming language, Java Persistence API (JPA)[7] and Hibernate Spatial[8] for geographic data object/relational mapping.

Figure 4 shows the architecture of the API, divided in three layers: *Domain Data Mapping*, *Handlers* and *Controller*. The *Domain Data Mapping* layer implements the database for the model of Figure 3, mapping Java objects into the underlying DBMS. The *Handlers* layer access the physical storage. This

[3]This paper is restricted to spatial aspects.
[4]http://code.google.com/p/dbv-ms-api
[5]http://postgis.refractions.net
[6]http://www.postgresql.org
[7]http://jcp.org/aboutJava/communityprocess/final/jsr317/index.html
[8]http://www.hibernatespatial.org

layer is inspired in the DAO (Data Access Object)[9] pattern, to which we added specific methods of our model. There are five handlers, each of which related to an entity of the *Domain Data Mapping* layer. The *Controller* layer is accessed by applications to select the DBV to use and to perform operations on.

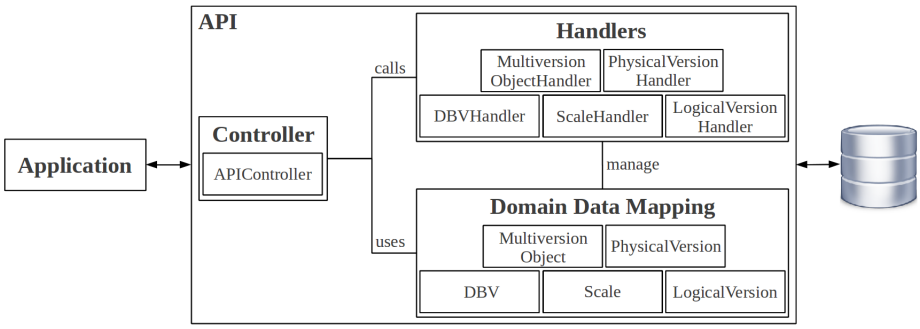


Figure 4: Architecture of the API

4.2 Using the API

This section shows the steps for an application A to use the API to create multi-scale databases.

Step 1 - Create subclasses of *PhysicalVersion*. First of all, it is necessary to create the schema, i.e., subclasses of *PhysicalVersion*, binding the appropriate parametrized type, which will indicate the *OBJTYPE* to be versioned. Figure 5 shows two examples of subclasses of *PhysicalVersion*. Both have the same attributes, but different versioned data (because of the different binding parametrized type). The *GeometryPV* subclass is versioning only the *spatialObject* attribute while *SpatialPV* subclass is entirely versioned. Each subclass created by the user will represent a different table in the multiversion database.

Step 2 - Add data. In order to add new data, the first step will be to select a specific version *stamp* of a DBV. Here, A inserts *multiversion objects* and their *physical versions*. A *MultiversionObject* class has three attributes: *oid*, *title* and a list of *PhysicalVersions*. The first is the identifier, the second is some title which identifies the object in the real world, and the third represents the associated *physical versions* plus their scales. Also, it is necessary to define the spatial scales to be available. Every time a *Scale* is added, a new derivation tree is created (by creating a root DBV). A *Scale* class has three attributes: *sid*, *type* and *value*. The first is the identifier, the second represents the type of the scale: spatial, temporal, etc, and the third attribute is the value associated to the *type*. For instance, for spatial scales, the *value* contains their size (e.g. 1:10000).

[9] http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

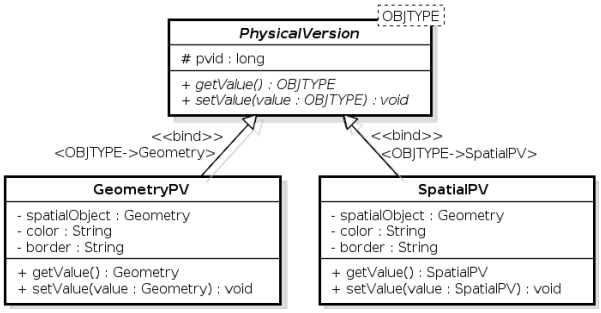


Figure 5: Examples of subclasses of *PhysicalVersion*

Step 3 - Perform operations. Once steps 1 and 2 are performed, applications can invoke operations on objects and DBVs, via invocation of methods of the *Controller* layer, e.g., adding, removing and updating the *logical versions*, by working in a scale at a time. Now, suppose we have already done the changes and we want to make a new version of them. When we create a new DBV from the current, the changes are saved. Subsequent versions can be built by changing the *logical versions* and creating new DBVs.

Consider Figure 6, where the roots (*stamp* 0) appear for scales 1:10000, 1:20000 and 1:50000. This example concerns urban vectorial data, and the Figure illustrates a given city section. The first version from root (*stamp* 0.1) shows the initial state of the section represented in the three scales. Version 0.1.1 and 0.1.2 show evolution alternatives in that section (either prioritizing the horizontal road, or the vertical road). The geometries in dotted lines represent the propagated data.

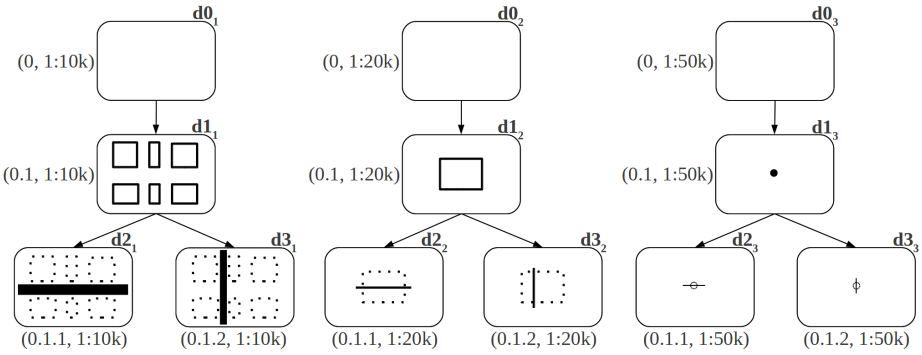


Figure 6: Multi-scale versioning problem example

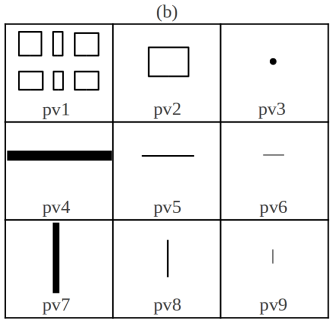
Internal details appear in Figure 7. Part (a) shows the *multiversion objects* and their *physical versions*. Part (b) shows the *physical versions* and their geometry. Finally, the *logical versions* and their relationship with *physical versions* are shown in part (c). For instance, in scale 1:10000, the city section is stored as

a complex geometry (a polygon with six sub-polygons), with *oid o1* and with three physical representations – one per scale – *pv1*, *pv2* and *pv3*. Each of these geometries will be accessible via a different DBV, respectively $d1_1$, $d1_2$ and $d1_3$.

(a)

MultiversionObject		
Title	**Oid**	**Physical Versions**
city section	o1	pv1 pv2 pv3
horizontal road	o2	pv4 pv5 pv6
vertical road	o3	pv7 pv8 pv9

(b)



(c)

LogicalVersion		
Multiversion Object	**DBV**	**Physical Version**
o1	$d1_1$	pv1
o1	$d1_2$	pv2
o1	$d1_3$	pv3
o2	$d2_1$	pv4
o2	$d2_2$	pv5
o2	$d2_3$	pv6
o3	$d3_1$	pv7
o3	$d3_2$	pv8
o3	$d3_3$	pv9

Figure 7: (a) *Multiversion objects* and their *physical versions*. (b) *Physical versions* and their geometry. (c) *Logical versions* from the example

Suppose the user wants to work at scale 1:10000, in the horizontal road situation, i.e., DBV $d2_1$. The DBV view is constructed from all objects explicitly assigned to it (*pv4* of *o2*), and all objects in previous DBVs of that scale, up to the root, i.e., $d1_1$ – *pv1* of *o1*. This construction of consistent scenarios for a given scale in time is achieved via the *stamps*, by the DBV mechanism. Notice that each version is stored only once. Unless objects change, their state is propagated through DBVs, saving space. Also, users can navigate across a path in the derivation tree, following the evolution of objects in time. For more details on space savings, see [CJ90].

5 Conclusions and Future Work

We have presented an approach to manage multi-scale geospatial data, and keep track of their evolution, through the DBV model. This proposal was implemented in a prototype, developed in order to validate our solution. We have already implemented some toy examples, which show the advantages of this proposal, and are constructing a test suite with real data.

Our framework supports the traceability of the evolution of spatial objects, while at the same time handling multi-scale management. Thanks to the adoption of the DBV model, storage space is saved [CJ90], and the separation between *physical* and *logical versions* facilitates the creation of consistent, single scale views over multi-scale data.

We point out that our approach is centered on data structures to store and manage multi-scale data. This allows controlling updates, keeping history of evolution in the real world and other issues that can be efficiently handled only in a storage based policy. Nevertheless, the DBV infrastructure can be

used as a basis for any kind of generalization approach – e.g., construction of intermediate scales, generalization of alternative virtual scenarios, and so on, to work, for instance in digital cartography.

Future work includes versioning along the temporal scale and specification of integrity constraints across scales, to determine rules for update propagation.

6 Acknowledgements

Work financed by FAPESP (grant 2011/14280-0) and CNPq (grant 133037/2011-8) and partially by the Microsoft Research FAPESP Virtual Institute (NavScales project), the Brazilian Institute for Web Sciences Research, CNPq (MuZOO project), PRONEX-FAPESP [10], CAPES (AMIB project), as well as individual grants from CNPq.

References

[BBB07] Y. Bédard, E. Bernier, and T. Badard. Multiple representation spatial databases and the concept of vuel. *Encyclopaedia in Geoinformatics, Hershey: Idea Group Publishing*, 2007.

[BPB10] D. Burghardt, I. Petzold, and M. Bobzien. Relation modelling within multiple representation databases and generalisation services. *The Cartographic Journal*, 47(3):238–249, 2010.

[CJ90] Wojciech Cellary and Geneviève Jomier. Consistency of versions in object-oriented databases. In *Proc. of the 16th Int. Conference on Very Large Databases*, pages 432–441. Morgan Kaufmann, 1990.

[DWL08] X. Deng, H. Wu, and D. Li. Mrdb approach for geospatial data revision. In *Proc. of SPIE, the Int. Society for Optical Engineering*, 2008.

[FJ03] A. Friis-Christensen and C. Jensen. Object-relational management of multiply represented geographic entities. In *Proc. 15th Int. Conference on Scientific and Statistical Database Management SSDBM*, 2003.

[GJ01] Stéphane Gançarski and Geneviève Jomier. A framework for programming multiversion databases. *Data Knowl. Eng.*, 36:29–53, 2001.

[GZH+10] Huijun Gao, Hao Zhang, Daosheng Hu, Ran Tian, and Dazhi Guo. Multi-scale features of urban planning spatial data. In *18th Int. Conference on Geoinformatics*, pages 1–7, 2010.

[10] Model and Methods in eScience for the Life and Agricultural Sciences

[Oos09] P. Oosterom. Research and development in geo-information generalisation and multiple representation. *Computers, Environment and Urban Systems*, 33(5):303–310, 2009.

[OS10] P. Oosterom and J. Stoter. 5d data modelling: Full integration of 2d/3d space, time and scale dimensions. In *Proc. GIScience 2010*, pages 310–324, 2010.

[PSVZ09] Christine Parent, Stefano Spaccapietra, Christelle Vangenot, and Esteban Zimányi. Multiple representation modeling. In *Encyclopedia of Database Systems*, pages 1844–1849. Springer US, 2009.

[PSZ06] C. Parent, S. Spaccapietra, and E. Zimányi. The murmur project: Modeling and querying multi-representation spatio-temporal databases. *Information Systems*, 31(8):733–769, 2006.

[RD07] Anne Ruas and Cécile Duchêne. Chapter 14 - a prototype generalisation system based on the multi-agent system paradigm. In *Generalisation of Geographic Information*, pages 269–284. Elsevier Science B.V., 2007.

[Sar07] L. Tiina Sarjakoski. Chapter 2 - conceptual models of generalisation and multiple representation. In *Generalisation of Geographic Information*, pages 11–35. Elsevier Science B.V., 2007.

[SPV00] Stefano Spaccapietra, Christine Parent, and Christelle Vangenot. Gis databases: From multiscale to multirepresentation. In *Abstraction, Reformulation, and Approximation*, volume 1864 of *Lecture Notes in Computer Science*, pages 57–70. Springer Berlin / Heidelberg, 2000.

[SVvO+11] Jantien Stoter, Thomas Visser, Peter van Oosterom, Wilko Quak, and Nico Bakker. A semantic-rich multi-scale information model for topography. *Int. Journal of Geographical Information Science*, 25(5):739–763, 2011.

[ZJ03] S. Zhou and C. B. Jones. A multirepresentation spatial data model. In *Proc. 8th Int. Symposium in Advances in Spatial and Temporal Databases – SSTD*, pages 394–411, 2003.