Spatial Database Consistency in Web Application Frameworks Case Study Django

Adam Kipkemei

March, 2010

Spatial Database Consistency in Web Application Frameworks Case Study Django

by

Adam Kipkemei

Thesis submitted to the International Institute for Geo-information Science and Earth Observation in partial fulfilment of the requirements for the degree in Master of Science in *GFM*.

Degree Assessment Board

Thesis advisor	Dr.Ir. R.A. de By Dr. J.M. Morales
Thesis examiners	Chair: Dr.Ir. R.L.G. Lemmens External Examiner: Dr. A. Wombacher

INTERNATIONAL INSTITUTE FOR GEO-INFORMATION SCIENCE AND EARTH OBSERVATION

ENSCHEDE, THE NETHERLANDS

Disclaimer

This document describes work undertaken as part of a programme of study at the International Institute for Geo-information Science and Earth Observation (ITC). All views and opinions expressed therein remain the sole responsibility of the author, and do not necessarily represent those of the institute.

Abstract

Web application frameworks promote the building of dynamic web applications based on the model-view-controller (MVC) architecture. The MVC architectural principle ensures that, a created web application complies with the 'Don't Repeat Yourself' (DRY) principle, is loosely coupled, is reusable, and is created in a rapid and clean manner. Web applications maintain implicit and explicit constraints either, fully in the application layer or partly in the database and application layer. Therefore, most web application frameworks compromise spatial database consistency because they lack sufficient capability to define explicit validation constraints in the application layer. Moreover, when constraints are maintained in the application layer, then a database shared by multiple applications is not exposed to the same validation standards. This is because different applications define the validation rules differently in their respective application layers, hence allowing flawed data to enter into the database. We investigated how web application frameworks are designed with regard to the extent at which spatial database consistency is maintained. After identifying the factors that influence spatial database consistency, we developed a constraint design method which was implemented using Django web application framework with GeoDjango. The constraints design method took into consideration the philosophy of maintaining constraints within the database layer and calling them into the web application using a Remote Procedure Call API, and observing taxonomic granularity of constraints during the design of integrity-preserving functions and data validation process. The design method also provide for the creation of update functions that call and perform the integrity-preserving functions. In addition, smart functions that do pre-processing can be embedded inside the update functions to add meaning to the stored data. We tested the design method and prototype design architecture using the Amazonian avian distribution data. Therefore, the prototype and constraint design method aims at maintaining spatial database consistency in web application frameworks.

Keywords

Django, spatial web application frameworks, spatial database consistency, constraints, GeoDjango, triggers, validation

Contents

List of Figures

List of Acronyms

API	Application Programming Interface
ACID	Atomicity Consistency Isolation Durability
CGI	Common Gateway Interface
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
DRY	Don't Repeat Yourself
DBMS	Database Management System
GDAL	Geospatial Data Abstraction Library
GEOS	Geometry Engine Open Source
GPS	Global Positioning System
GUI	Graphical User Interface
ER	Entity Relationship
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
MTV	Model Template View
MVC	Model View Controller
TIN	Triangulated Irregular Networks
OCL	Object Constraint Language
OGC	Open Geospatial Consortium
ORDBMS	Object-Relational Database Management System
ORM	Object Relational Mapping
UML	Unified Modeling Language
WML	Web Modeling Language

XML Extensible Markup Language

Acknowledgements

It is with my deepest gratitude that I would wish to express my appreciation to my research project supervisors Dr. R.A. de By and Dr. J.M. Morales for their persistent guidance and assistance during the entire research study period. Their invaluable assistance during the proposal writing, proposal defense, midterm presentation and the final thesis writing and final defense preparation. I appreciate their wealth of knowledge in prescribing literature and giving well informed advice that shed light especially when I was stuck or experienced hardship in deciphering technical concepts.

I also would like to thank Wan Bakx for helping me to develop my talent in running besides my studies in ITC. I would also like to thank my fellow GFM classmate for their help and day to day interaction.

Finally, I would like to thank my family for being understanding when I was far from home and their words of encouragement.

Chapter 1

Introduction

1.1 Motivation and problem statement

Since its inception, the Internet has undergone tremendous growth and development in quick dissemination and use of content and services across the web global village. Initially, the Internet was made up of static web pages that displayed contents like text and images [14]. This enabled the users to only navigate through the web pages without adding any of their own inputs. However, with the advancement in web technology, desktop applications were made possible online as web-based application [14]. This gave rise to interactive web application christened web2.0 (wisdom web) applications [22]. These applications have enabled users to add their inputs into the web application which are later stored in back-end databases.

Web application frameworks have been used to build web 2.0 type of application in a clean, rapid and reusable way [19]. This is because they are based on the model-view-controller (MVC) architecture that splits the application into three levels; model, view and controller as shown in Figure 1.1 [31]. A model represents an application's data and contains the logic for accessing and manipulating that data. The view is responsible for rendering the state of the model. The controller is responsible for intercepting and translating user input into actions to be performed by the model [31, 14]. There is a clean logical separation of the view design and the code beneath partial classes [31]. The controller handles centrally all client requests. This promotes a cleaner division of responsibilities for the controller that normally deals with view and navigation management [31]. The model access and manipulation roles are left to request handlers, which are request-specific [31].

The view is coded in a loosely coupled way that separates it into physical assemblies enabling code re-use within application and in other applications, without breaking other layers of code [8]. This saves greatly on development and maintenance costs and avoids code replication. The DRY "Don't Repeat Yourself"principle makes the code clean (no spaghetti) [8]. That is, if a configuration is made in one web page it does not need to be replicated in all the pages. Common web application frameworks include TurboGears, Ruby on Rails, Django, GeoDjango, Struts and Web2Py [8, 14].

Conceptual database design has been a classical architectural method in

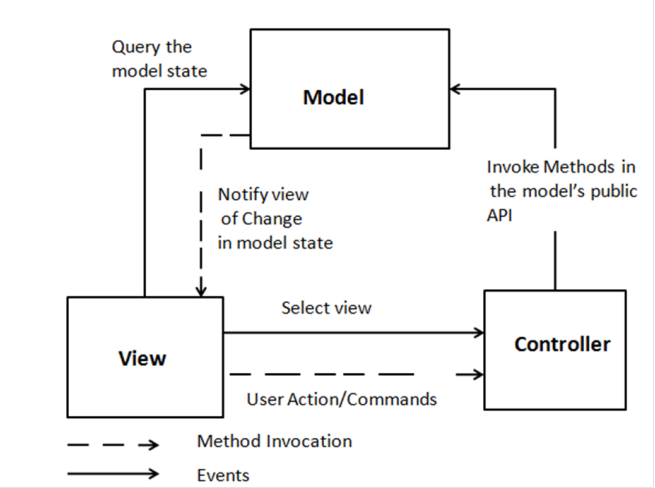

Figure 1.1: Model-View-Controller Architecture (Adapted from (27))

designing spatially consistent back-end databases accessed by web user interfaces [7]. This is realized through the use of modeling tools like Unified Modeling Language(UML) and its subset called the Object Constraint Language (OCL) [26]. They are used to model the database in a "bottom-up "approach, taking into account class cardinalities, spatial constraints and many integrity constraints [5, 26]. All the database constraints are maintained fully in the back-end database. Hence, there is clear separation of the web application design and the database accessed. Therefore, this spatial database is easily plugged and played into any application. Modeling of the spatial database starts with a conceptual design then logical design, which is finally translated into physical database [7]. For instance, if a spatial database is built in PostgreSQL, which is an open source spatial database that supports OGC (Open Geospatial Consortium), simple feature spatial data types [4], there will be maintenance of spatial database consistency within the PostgreSQL database.

However, a typical web application framework uses a model-design to build databases that maintain some standard consistency. Model design uses an object-relational mapping (ORM) library to create dynamic database and query through object interface [8]. Therefore, only standard consistency rules are maintained, and they are maintained in part within the web application, not within the database. Hence, the database lacks spatial and other forms of consistency if the framework adheres to its standard constraints, neglecting the OGC standards spatial constraints. It's also difficult to plug and run this database into another application without breaking the logical functions, because standard consistency is maintained partly within the web application [31]. Moreover, lack of standards for integrating different web application frameworks jeopardizes efforts to create and share spatially consistent databases amongst web applications created using different types of web application frameworks [15].

Taking Django as our case study, the research will be conducted to come up

with a design method that harnesses the strengths of MVC architecture like DRY principle and conceptual database design approach. A prototype framework will be designed to implement the design method that considers maintenance of spatial consistency fully in the database and enforces constraints based on bottom-up approach based on taxonomic granularity of data structure. Thereafter, a web application will be built based on a given use case to test and evaluate whether spatial database consistency was fully maintained within the database and also check whether the strengths of MVC architecture were integrated. This will result in an independent, fully-fledged spatially consistent database that can be plugged into and used in any application. This needs thorough research in web application design and modeling because a mere combination of view, model database design and classical database design in an ad-hoc way without following a method, will not necessarily result in a robust implementation.

1.2 Research identification

The research project aims at studying the spatial database consistency in web application frameworks that help in building rapidly clean dynamic web applications.

1.2.1 Research objectives

1. To review and understand the architecture and functionality of web application frameworks.
2. To determine the degree at which web frameworks maintain spatial database consistency.
3. To define a design method that incorporates maintenance of spatial database consistency.
4. To build a prototype web application framework that implements the method designed to incorporate maintenance of spatial database consistency.
5. To test and compare the performance of prototype web application framework against the performance of Django web framework with regard to maintenance of spatial database consistency.

1.2.2 Research questions

1. What are the fundamental principles of web application frameworks?
2. To what extent is spatial database consistency maintained in a Django web application framework?
3. What design method is suitable to employ to introduce spatial database consistency in Django web application framework?

4. Which part of the Django web application framework code can be extended to support database consistency?

5. Does the prototype web application framework work better in terms of maintaining spatial database consistency than the Django Web application framework?

1.2.3 Innovation aimed at

The research is geared towards designing a constraint method and implementing it by extending Django web application framework to support remote call of integrity-preserving functions that enforce constraints in bottom-up approach based on taxonomy of data structure granularity, and also ensuring that all consistency constraints reside in the application layer and not the database layer. This design will pave way for smart databases that applications will not only view databases as repository to store spatial and attribute data but also put predefined meaning into data by enforcing constraints and other business rules before data storage. This will result in a prototype web application framework that maintains spatial database consistency within the database while still maintaining the strengths of MVC architecture.

1.3 Project set-up

We enumerate how we intend to tackle the research problems to achieve the set objectives.

1.4 Method adopted

We adopted waterfall design model applied in software development process. It is basically split into five main facets that include: requirements analysis, design, implementation, verification and maintenance [30].

We delved into the working mechanism of major existing web application frameworks, that is, Django, Pylons, GeoDjango, Web2py and TurboGears. We studied the database building mechanism to understand how the main system tenets handle spatial database consistency. Thereafter, we came up with system requirements to be improved. We also studied UML and WebML to see how these modeling tools could be harnessed in building database consistency in web application frameworks. At the end of process, we came up with design factors that need to be considered to attain spatial database consistency in web application frameworks.

After identifying what needed to be improved, we developed constraint design methods and an architectural design that helped in solving the problems of spatial database consistency. This design methods incorporated two research philosophies of maintenance of all constraints at the database layer and developing an integrity-preserving functions that enforced constraints based on taxonomic granularity of data structure in a bottom-up approach. We later studied

the Django code and see how to improve it in a manner that will fit like dovetail joint with the created module extension library, to come up with a prototype web application framework implementation. A module extension added using the Python programming language, was integrated into the Django web application framework. The functionality was aimed at realizing spatial database consistency.

We later tested our designed implementation by building a web GIS application of Amazonia avian distribution, that is a dynamic Web 2.0 application. The provided Amazonia avian distribution datasets were used to test and evaluate whether the designed prototype web application that supported validation of datasets using by passing supplied data in integrity-preserving functions improved the maintenance of spatial database consistency in web application frameworks.

1.5 Outline of the thesis

Chapter 1 is an introduction to the research work, it discusses the motivation and points out the problem statement. Furthermore, the research objectives are stated and related questions are raised. Finally, the method adopted is elucidated.

Chapter 2 primarily ventures into the emergence, growth and development of the web application frameworks and the technology behind them. The MVC architecture the enables rapid creation of websites is explained. Spatial databases aspects that are the kernel of the research are outlined and explained in details. Django as the framework of choice for the research study is systematically expounded.

In-depth study is further conducted to understand how integrity constraints of various natures can be adopted and modeled into Django, to support creation of integrity-preserving functions that yield spatially consistent databases.

Chapter 3 dives into the design aspect to be observed in order to instill spatial database consistency. In the design, we take into consideration research the project's two philosophies of maintaining constraints in the database layer rather than the application layer and observing taxonomic granularity of constraints during data validation. Moreover, the user and application requirements as well as the requirement analysis considered in the design method are discussed.

Chapter 4 focuses on the execution of the design through implementation of the design method. This involved re-engineering and extending Django to allow spatial consistency constraints to be created into the database by Django to guard the integrity of the spatial database. The prototype Django web application framework is tested using an Amazonian avian distribution use case to evaluate to what extent it has been able to maintain spatial database consistency.

Chapter 5 concludes the research by giving a summary of the results by discussing the web applications performance built by Django and the prototype web frameworks. The strengths as well as the limitations of the prototype

framework performance are discussed. Moreover, further improvement leads for future investigation are pointed out and new questions arise as well. Finally, the thesis is concluded.

Chapter 2

Principles of Databases and Web Application Frameworks

2.1 Web 2.0 introduction

The World Wide Web has grown astronomically over the years since its inception. The web revolutionary impact, its massive extent and outreach has tremendously changed each level of our society. This is evident from the way the web has altered the means to interact, participate, transact, gather and disseminate information. It has brought the world together and made it a small village [16].

The fundamental goal for development of the World Wide Web was to enable easy access of information by users distributed across the globe in a consistent way [14]. The HTTP protocol was developed to enable the exchange of information requested by client computers from server computers. Therefore, the web was regarded as a vast repository of static information. Initially, the web was mainly used by the scientific community but with development and affordability of desktops, Internet services became a household product [25].

2.1.1 Trend

The web has undergone a paradigm shift from a general repository for storage and retrieval of static pages to a powerful infrastructure for developing and running sophisticated dynamic web applications called web 2.0 (wisdom web) [25]. The emergence of new technology like dynamic tools and languages, together with innovative methodologies has made Web 2.0 applications come to fruition [14]. Initially, dynamic web applications were programmed using the Common Gateway Interface (CGI) [8]. But CGI had a number of drawbacks, like limited reusability and extensibility [3]. It was also cumbersome for the server to manage because all code resided in the server [14]. This resulted in a desire for a standard language with agile development methods hence the creation of web application frameworks. The web application frameworks promote the separation of concerns through their MVC architectural design. For instance, the code that manages the data (i.e. model) is separated from code that is responsible for rendering data (i.e. view) as well as functions that

move data from component to component (i.e. controller) [8]. Moreover, the client script that interacts with the user runs on the client-side (i.e. browser) while the server-side script that performs the server processes interacts with the database, running the server [14, 13].

Web 2.0 applications are dynamic web based applications that are executed by web browsers via the Internet [14]. Web 2.0 technology has changed the role of the user from a passive spectator to an actor. This has enabled users to interact with the web application, that is, they can add their own inputs interactively [25]. This has greatly shifted the software development to include user participation. That is, front-end web applications interact with connected distributed databases.

2.1.2 Web Framework Technology employed to build web 2.0 applications

Web 2.0 applications use dynamic tools and languages that provide an appropriate development model for a rapid release cycle, debugging and quick responses and deployment to user needs [20]. Therefore, dynamic languages allow information to easily flow, enabling dynamic content. Compiled static languages take a long time to deploy and are more complicated during deployment [20, 8]. Most Web 2.0 applications run on platform-independent dynamic programming languages like Ruby, PHP and Python [20]. Common web frameworks based on these languages that are used to build web 2.0 applications include Django, Ruby on Rails, Struts, Web2py, Turbo Gears and Pylons [8]. These innovative web frameworks automate the tedious repetitive processes of building a web application by generating code, and hence leave the developer with more time to be creative [3]. Most of web frameworks' bundled libraries operate on the MVC (Model-View-Controller) architectural design [31, 8].

2.1.3 The benefits of Web 2.0

- The web application release rate is faster than the traditional product release cycle for desktop applications [20]. This is because of the suitable development methodology adopted that allows quick fixes of bugs which are often quickly spotted and reported by the wide range of users. It takes as short as two weeks to have new updated functionality without a need to install updates; its update is all done centrally by the developer [20].

- It involves users as co-developers as well as real time application evaluators. The web exposes the web application to a large number of users with diverse user need. Hence, the developers can monitor the system's performance. This presents the software to the users' environment, which provides a concise model to evaluate it. With this capability, the developers are much removed from the traditional laborious way of using prototype releases to be tested basically by a few [20].

- New products are created because of quick releases and much feedback from the users.

- Increased responsiveness.
- Web 2.0 has enabled users to interact with the database making it richer.

2.1.4 Impact of Internet web applications to desktop applications

In the era of Internet advancement, Web 2.0 has transformed software from package software that came in versions and needed updates to web applications that are viewed as ever available services by users [20]. The users do not mind about what version an application is because the developer keeps updating it over and over with time. Therefore, there is no need for versions, updates and installations. This has brought to an end the traditional packaged software adoption cycle of design-develop-test-ship-install [20]. Instead, it has been replaced by the notion of software being a service that is ever improving [20].

2.2 MVC architecture

The Model-View-Control (MVC) [34] design paradigm allows the separation of the code that controls business logic and application data from the code that manipulates presentation data to the user and event handling [8]. The act of separating the web application into Model-View-Control, the three logical components, simplifies web application development and maintenance [31]. Originally, the development of MVC was to map the traditional input, processing, output roles of a Graphical User Interface (GUI) environment as shown in Figure 2.1. However, with advancement it developed into a complex architectural design pattern with sophisticated roles amongst the model, view and controller triad as shown in Figure 2.2.

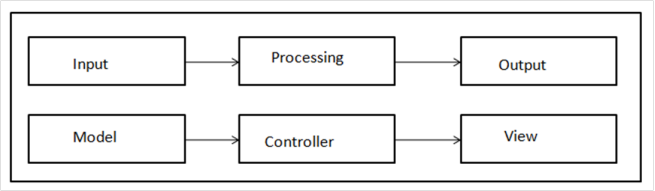

Figure 2.1: Initial basic MVC process (Adapted from (1))

2.2.1 Model

The model layer represents the application data and is also responsible for the business logic of accessing and updating the data. The data facets that constitute a continuous state of the application are stored and managed in the model objects [31]. The model manages data elements and notifies observers when that data changes. A model hosts data and functions related by a common purpose, which later translates into a table schema in the physical database [31].

Therefore, for unrelated data and functions to be modeled, two separate models have to be created.

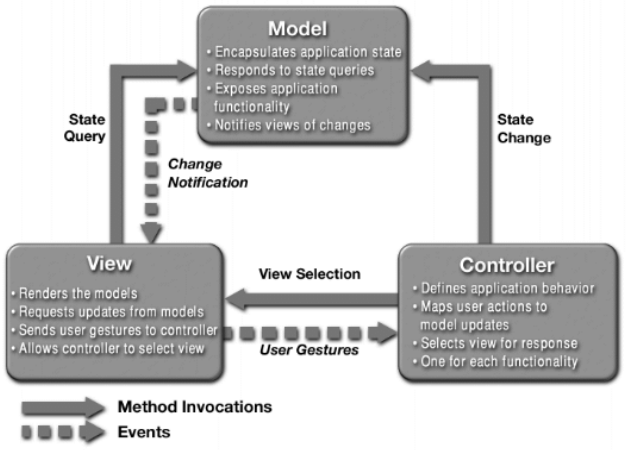

Figure 2.2: MVC Architecture (27)

A group of data and operations within the model wraps and abstracts the functionality of the modeled business process. The model encapsulates access to the data stored and provides a reusable class library of functionality [34]. The model not only encapsulates the data and the functions that operate on it, but also serves as an abstraction or approximation of some real world system or process [8]. For instance, a model defined to bridge the back-end computation and the front-end GUI presentation, will gather appropriate data from the database and deliver it to the user's GUI in a way that the users can relate with. Within the model, reside the operations like database abstraction, authentication and validation rules [34]. Some of the model functions to data include store, retrieve, emulate, convert, sort and convert amongst other operations.

Methods for accessing, updating and executing complex encapsulated processes inside the model are exposed by the model's interface. A model defines how data is accessed and set for a given page, including any functions necessary for security or data validation and modification [31, 34] . The services rendered by the model should be generic, to support a wide variety of clients being served. The services rendered by the model are accessed by the controller through database querying, or respond to instructions to change the state of the model [34]. The model does not have a visual representation, hence the model is not aware about how the data it contains will be presented and displayed within the user context, that is browser.

2.2.2 View

The view layer plays a key role of rendering the state of the model on the display surface to the user [8]. The view manages the area of display, ensuring presentation of data to the user as a combination of graphics and text. The semantics of presentation are encapsulated inside the view, hence this enables

the modeled data to be adapted by numerous types of client [31, 34]. The view is dynamic because it modifies itself automatically to reflect and maintain consistency of the communicated changes within the model [34]. In addition, multiple views can render simultaneously the contents of the model to a wide variety of display surfaces. Moreover, the view is a platform for forwarding user input to the receiving controller [31]. Typically, the view layer plays the role of web page rendering, that controls the look and feel of application and facilitate data collection from users via view technology like HTML, CSS and JavaScript [31].

2.2.3 Controller

The controller layer is in charge of intercepting and translating keyed-in user input into actions to be executed by the model [31, 34]. Users use the controller as a means to interact with the application, this is because the controller accepts user input and instructs the model and view to execute input based actions. That is, the controller calls model facilities and interprets the incoming data which is rendered by the view. Therefore, the controller determines the next view based on user input and also the model operation output [31]. Hence, the controller maps end-user actions into application response. Some of the user inputs accommodated by the controller include: HTTP requests from web clients, WML from mobile clients and XML-based files. Finally, the controller layer role is to take charge of the application exception (errors) and flow control, hence it glues and merges the view styling together with the functionality of the model [8].

2.2.4 How MVC operates

The model, view and controller are mutually related and are in constant contact, hence it is mandatory that they reference each other [34]. The relationships of the Model-View-Controller is illustrated in Figure 2.2.

Figure 2.2 above portrays the fundamental lines of communication amongst the model, view and controller. To begin, the model points to the view, this implies that the model sends the view notifications of change as indicated by the broken line pointer. Moreover, the model's view pointer is just a base class pointer; the model should not be aware of the kind of view that observes it [34]. In contrast, the view is aware exactly of the kind of model observing it. The view has a continuous line pointer to the model; this indicates that the view is allowed to call any of the model's functions. In addition, the view also has a pointer to the controller; nonetheless it should not call functions from the controller besides those defined in the base class [34]. This is primarily to keep the dependencies minimal in case one intends to swap out one controller in place of another. In summary, model and controller communicate with the view via events indicated by broken line pointers. Notice as depicted by continuous line pointers that the controller has pointers to both the model and the view, and therefore it knows the kind of the view and the model. Finally, since the behavior of the triad (model-view-controller) is defined by the controller, it must know the type of both the view and the model so that it translates the user input

into the required application response [8, 34].

2.2.5 Object Relational Mapping (ORM)

ORM is a collection of libraries in a web framework that maps the SQL database columns and tables onto objects and classes within a model [21]. This enables the classes and the instantiated objects to be used in a normal programming language like Python, and methods defined to the behaviour of the classes, that is the tables. The ORM is concerned with mapping objects to a relational model, so that the objects (created in a given programming language) become persistent in a given RDBMS [29, 33]. Figure 2.2 shows ORM as a liaison between application and the database. An object that is persistent can automatically store as well as retrieve itself in permanent storage [33]. Whereas, an RDBMS stores only data, the objects have identity, state and behaviour together with data.

Unlike writing data directly into the database using a data-aware GUI application, using objects results in benefits of encapsulation [29]. Encapsulation is a signature of loose coupling because it enables interaction of objects without knowledge of implementation details. Loosely coupled applications are easy to maintain, because they lack the cascading effect when code is modified. However, objects can't be directly saved into and retrieved from a relational database, hence the need for an object-relational mapping. Relational modeling aims at normalizing data to remove table data redundancy whereas the object-oriented design approach aims at modeling business logic by coming up with real-world objects having data and behavioural traits [21].

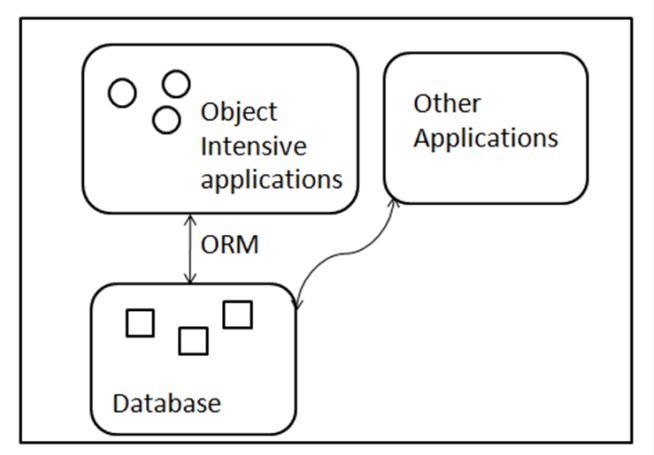

Figure 2.3: ORM used by object intensive applications (Adapted from (2))

ORM tools therefore strategically map objects into relations to harness the benefits of relations and objects [21]. Web application Frameworks based on the MVC architecture use ORM to create a database schema indirectly by translating objects into SQL statements that are later used to create tables in a database [8]. However, in our design we aim at harnessing object strengths

as well as making it optional to either create schema from ORM or from the database directly. However, the fundamental goal will be to maintain spatial consistency wholly in the database.

2.2.6 Advantages of MVC architecture

The MVC architecture has a number of advantages that makes it a desirable design pattern for building web applications. The advantages are a result of decoupling business logic, data access, and data presentation and interaction by users.

1. Web applications built on MVC architecture are reusable, since they are decoupled they depend on few classes that are easy to reuse. Therefore there is no need to invent new solutions to recurring problems, developers only need to follow the pattern and adapt it accordingly.

2. The MVC design pattern simplifies web application development. This is because most of the code are auto-generated by the web application framework.

3. MVC based web applications are loosely coupled, based on a separation of concerns.

4. The DRY ‘ Don’t Repeat Yourself’ principle ensures clean and easy to maintain code because of central configuration of pages.

5. Decoupling results in independent components of a web application which makes them easy to maintain. This is because there is no ripple effect across the components when one part of the component is changed.

2.3 Top-down versus Bottom-up database design

There exist two common methods of designing a database, the top-down approach and the bottom-up approach as shown in Figure 2.4. The two design approaches aim at achieving database responsibilities, that is to store, modify and retrieve data.

2.3.1 Top-down approach

In the top-down approach, the database administrator starts with the general view and then systematically moves to specific details as shown in Figure 2.4 [28]. The developer will build the general parts of the system and later consult with the end-users on what needs to be integrated into the database system. This implies that the developer will work together with the users to define the final database. To end up with an effective database, the developer has to possess a deep understanding of the system. Web application frameworks tend to mostly use this approach in the design of models that create the database. This top-down approach tends to be error-prone. This is a drawback

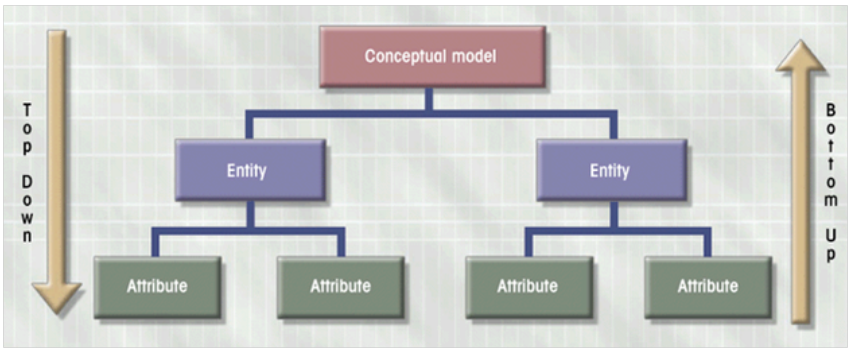

Figure 2.4: Top-down-Bottom-up approach to database design (Adapted from (24))

in designing consistent databases, especially if the user and developer miss to incorporate vital details.

2.3.2 Bottom-up approach

In this approach, the database developer begins with the specific details like the object classes, relations, business rules, user interface before venturing up into the conceptual design as shown in Figure 2.4 [28]. The developer will then work backward over the system to determine what data needs to be stored in the database. It is good database design practice to first conduct a conceptual design before getting to physical design, reduce omission and errors. A number of modeling languages like CASE and UML are available to assist in drawing up Entity Relationship (ER) diagrams as class objects together with their respective associations, multiplicities and cardinalities [26]. Complex system requirements can well be modelled using this approach.

2.3.3 Classical database design

Classical database design mainly focuses more on the data structure of the database than on database functionality. Hence, we have the conceptual database schema which is defined as data content in a data model, independent of platform. The schema constitutes the class of objects, constraints, relation and multiplicity. This can be modelled using standard modeling tools like UML or WebML [26]. The modelled conceptual design output is portrayed in a graphical way that is easy to interpret and understand, as shown in Figure 2.5.

The conceptual design is then translated into a logical design by the modeling tool. The logical design is a composition of what the business logic is, and it models the entire operation process. Finally, the logical design is ultimately translated into a physical database, which is the end-product [26]. At this stage, the database is mostly not ready to be used because some business logic cannot be defined easily and modeled by the modeling tool. Therefore, this has to be defined explicitly by a database developer using check constraints together with functions and triggers.

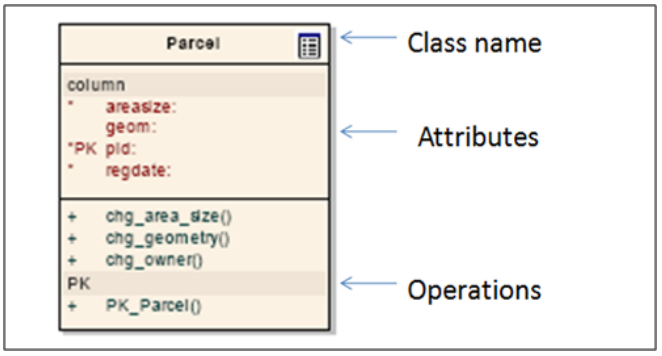

Figure 2.5: Class diagram drawn in a modeling language

2.4 Integrity constraints

After the schema is defined, integrity constraints are defined before data entry. Whenever the state of the database is to change, these constraints are first fulfilled before a database transaction becomes complete. When no error is raised, the data within the database represents an intended modeled reality . Integrity constraints can be categorized into two major classification approaches, that is based on DBMS and taxonomic granularity [9].

2.4.1 Constraint categorization based on DBMS support capabilities

This classification approach is further split into implicit and explicit integrity constraints.

Implicit integrity constraints

They are the basic inherent constraints support by the DBMS that manages the database operations. They include referential integrity, record/domain constraint, and key integrity [9].

Explicit integrity constraints

They are integrity constraints that are defined by the database designer to meet the user-defined business logic. They can be expressed as triggers, functions and check constraints [9].

2.4.2 Constraints categorization based on data structure granularity taxonomy

They are integrity constraints based on granularity of data structure storage in the database. There are four granularity primitives in total, that is attribute constraints, object constraints, table constraints and database constraints [9].

In this research project, we intend to use this approach as part of the checklist towards spatial database consistency.

The underlying principle is to determine a minimal amount of data within the database that needs to be validated, to check whether it violates or meets the prescribed integrity constraint [9].

Attribute constraints

These are integrity control rules imposed on individual attribute values pegged on a given condition. For instance, the name attribute value must be 'a string that begins with a capital letter'.

Object/Tuple constraints

These are integrity rules imposed on a combination of attribute values within a given individual tuple. For instance 'the date of birth cannot be greater than the date of enrollment into the army'.

Table/Relation constraints

These are integrity rules imposed on an array of tuples within a given table. For instance 'a sales territory table should not have more than 15 sales territories'.

Database constraints

These are integrity rules applicable to at least two different tables. For instance, based on related table by foreign keys akin 'Entry x in table A will be entered if and only if it exist in table B'.

2.5 Database consistency enforcement based on granularity

Database consistency, in this approach, is enforced in a bottom-up fashion, based on the granularity of the data. The bottom base being object consistency, while the middle part is made up of table consistency, and the topmost being the database consistency [9]. Upon enforcing the uppermost database consistency, it will have inherently constituted the embedded preluded object consistencies and table consistencies [9]. Figure 2.6 shows the database consistency based on taxonomic granularity.

2.5.1 Object consistency (Bottom level base)

Object consistency constitutes the base foundation for building database consistency bottom-up [9]. At this level, a finer list of object constraints and tuple constraints are enforced, paving way for higher level constraints before the state of the database changes during a transaction.

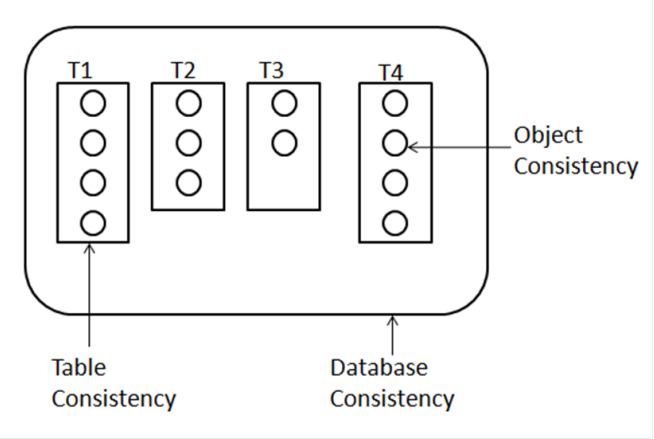

Figure 2.6: Database consistency based on granularity (Adapted from (9))

Object consistency is explained in the expression below [9]. The object universe class for a class C, is the collection instances objects t of type τ that obey all the rules ϕ for the class.

$U_c = \{t : \tau | \phi(t)\}$

Where:

- t (a tuple variable) represents an object state.
- τ depicts the record format of objects of class C and lists attributes for that class, and including their respective data types.
- ϕ represents all the constraints enforced on objects of class C. This includes the attribute constraints acting on an attribute and the object constraints between attributes within the same object.

  The class universe U_c is the infinite collection of allowable objects for class C.

2.5.2 Table consistency (midway level)

Table consistency is the midway stage in building consistent a database [9]. At this stage, a list of table constraints is enforced in addition to prior constraints captured in object consistency.

Table consistency is portrayed in the expression below. The class universe U_C is known, the table universe for class C T_C is defined, for each class C.

$T_c = \{tbl : \mathbb{P}\tau | tbl \subseteq U_c \wedge \Phi(tbl)\}$

We have here:

- *tbl* (a variable) represents a table state as a set of objects of the class C.
- All objects in *tbl* must be consistent to qualify as a member of the class universe U_C, the first condition.

- Besides the constraints enforced at the consistency stage, additional constraints that are needed are given in $\Phi(tbl)$. This includes, key uniqueness amongst other sets of explicit constraint enforced on the table level like check constraints and functions.

T_c is the collection of allowable table states for class C.

2.5.3 Database consistency (Top level)

Database consistency is the final stage in building a consistent database [9]. At this stage, a list of coarse granularity database constraints is enforced in addition to prior constraints captured in object consistency and table consistency stages. Once the database consistency is fulfilled, all prior constraints are inherently fulfilled guaranteeing integrity of data as the status of database changes after a transaction executes. Hence, once all these conditions are fulfilled the atomic data is updated, inserted or deleted from the database.

Database consistency is captured in detail in the expression below [9]. With the knowledge of class and table universes, the database universe DB is defined.

$DB = \{tbl_1 : \mathbb{P}\tau_1 ....., tbl_n : \mathbb{P}\tau_n | tbl_1 \in Tc_1 \wedge ..... \wedge tbl_n \in Tc_n \wedge \Psi(tbl_1 ...., tbl_n)\}$

Where:

- tbl_i (a table variable)represents the table for class C_i.
- A consistent database is at least a consistent collection of table states.
- The database constraint Φ captures intertable integrity rules like referential integrity.

2.5.4 Spatial taxonomic granularity constraints formulation

The examples below show various granularity primitive levels of spatial constraints as expressed in logical statement. These constraints are input in an integrity-preserving function stepwise for each granularity when building the business logic.

Spatial object constraints

The constraints enforce a rule that ensures that a sales zone geometry does not have a hole.

$\forall s \in Saleszone : numInteriorRings(s.geometry) = 0$

Spatial table constraints

The constraint enforces a rule at the table that ensures all parcels make up a contiguous area. It does so by looking within the table's topology to ensure that adjacency is maintained. If a geometric feature that is not adjacent to any other is entered it will be rejected by the database integrity control.

$\forall l, m \in Saleszone : l \neq m \Rightarrow Disjoint(l.geometry, m.geometry) \vee Touches (l.geometry, m.geometry)$

Spatial database constraints

These constraints enforce a list of rules at the database level (across-tables) that ensure sales zones geometry stored in table Saleszone are mutually disjoint and each sales zone falls inside its sales region that is stored in table named Salesregion. To achieve this, the Saleregion table is referenced and checked before data is stored in the Salazones table, once the conditions are met.

$\forall l \in Saleszone, f \in Salesregion : (l, f) \in PartOf \Rightarrow Within(l.geom, f.geom)$

2.6 Database consistency maintenance in Web 2.0 applications

Consistency validation in web applications is enforced mainly in two layers, that is, at the database layer and at the application level [8]. Consistency is maintained at the application layer when a web developer codes the constraints at the web application that is connected to a database. When a web application framework is being used to build a dynamic application the controller provides support for definition business rules functions that reside in the web application. On the other hand, consistency is maintained at the database layer when database validation rules are defined in the database by a database administrator who ensures all the rules reside within the designed database.

2.6.1 Application level integrity control

Application level integrity control is when the web UI is used to encode business logic in forms/screen GUI/web pages [23, 6]. They are coded using client scripts. The client script can duplicate the logic encoded in the server side. This client sided script cannot be trusted because the script can be turned off or a form entry field manually manipulated. The database consistency maintenance at application level should be avoided because it carries with itself limitations like the database being exposed to different constraints if its is accessed by multiple applications having differently programmed business logic, this can compromise the integrity of the data stored in the database.

Most web application framework ORMs have limited capabilities to wrap and map into the database all the integrity constraints defined. Only a couple of inherent database constraints are mapped into the database by the ORM [8]. Such constraints include primary keys, reference keys, domain types and multiplicities [12]. Most of the business logic is defined within the web applications mainly at the controller and the view (user interface). This is due to some ORM inability to support server-side programming languages like PL/pgSQL and PL/Java. Therefore, integrity enforcing operations supported by databases like check constraints, functions and triggers are not supported by web application frameworks. Hence, all the business rules are captured in the web application itself. For instance, the Ruby on Rails web application framework views the database as a dumping repository for storing data only [6]. As a result, Ruby on Rails maintains entirely all integrity constraints within the web application framework. Django, for instance, does not support definition of check

constraints and triggers. The MVC architecture provides for validation of data either solely at the application layer or double validation at the application layer and the database layer.

2.6.2 Database level integrity control

A database should be responsible at all cost for the validity of data input data since an inconsistent database loses its role as an effective and reliable system for storage and retrieval of data. Spatial database consistency is controlled within the database by ensuring that all constraints reside within the database wholly, whether defined at application level (e.g., by the ORM) or defined physically at the database level. The web framework is only notified of the existence of a constraint in the database through an API. This constitutes part of database security as a whole. Databases are primarily designed to handle constraints better than the client applications, this is one of database's main roles.

The enforcement at the spatial database level is enabled by employing functions and triggers [11]. This enables the definition of complex spatial rules using PL/pgSQL procedural language that is supported by the PostgreSQL database, together with built-in predefined PostGIS functions [9]. Both simple and complex spatial business rules operations can be handled well at the database level by use of procedural server-side programming [11]. The transaction functions will encapsulate the data submitted to the database to ensure users do not enter data directly into the database for integrity reasons.

When basic constraints are not stored in the database, at one point invalid data might slip into the database due to application bugs and may raise exceptions during querying or report generation.

2.6.3 Strengths of maintaining consistency at the database level against the application level

We believe strengths of integrity constraint controls maintained at the database layer, are as follows.

- All the applications data entry in the database are exposed to the same level of data quality checks, because of the central enforcement of the integrity control within the database to restrict what gets into it.

- With multiple applications possibly written in different languages, running in different platforms putting constraints in the database layer strengthens and protects the database. Since having two different applications, the chance of having similar input data validation controls is remote.

- It is easier for the designer of the application to focus on the application development without worrying about check constraints at web user interface (UI) point of data entry since the constraints exist where data is finally stored in the database.

- It enhances flexibility in application design to meet frequently changing demands for new UI. That is, if the database caters for constraints, the front-end developer is at liberty to design an application that hooks onto the database without worrying about back-end consistency.
- In an environment where an application is maintained by a large pool of developers, implementing the database constraints in the database eliminates possible disaster when changes of integrity controls are made at the application layer.
- When building databases that store data received from multiple applications, in which the database administrator has no control of the application programming, maintaining constraints at the database level will validate and intercept erroneous data entered through broken applications.
- It avoids duplication of constraints at the database and application level. Hence, it avoids unnecessary redundancy that might burden the system's processing and make also make tracing of errors difficult.
- It works well for new web applications that are embedded in legacy databases that implement business logic at the database layer. If the legacy database relies on constraints at application layer it involves much effort for developers in understanding and translating correctly the business rules implemented at application layer. Legacy databases with constraints fully maintained a priory at database layer come with a tested validation mechanism that makes continuation easy.
- Enforcing integrity control at database layer makes the integrity-preserving functions easily reusable, that is, any web application can call the functions stored in the database and use them to insert data into the database without having to build business logic from scratch.

2.6.4 Limitations of database consistency maintained at the database level

Inasmuch as consistency maintained at the database level has a number of strengths, it also has a number of limitations, as follows:

- Triggers make the database opaque, that is they are not visible to the user interacting with the database.
- Overloading the database with foreign keys may overload it. This is because the database integrity control has to check across tables before a given data entry is validated.

2.7 Addressing the problem of consistency management in database level

To ensure that the consistency is fully maintained within the database, all the business logic specifications must be upheld explicitly by check constraints,

functions and triggers within the database. The web application will call these functions when there is any update, insertion and deletion of data within the database.

The database developer is charged with full responsibility to ensure s/he has control over database consistency; this is attained by ensuring that all integrity controls reside within the database. The administrator ensures that the business logic is built once for front-end user applications within the back-end database as a good practice using functions and views [9]. This ensures that the front-end users update and modify data uniformly and see the same view of data.

2.7.1 Spatial databases design

Spatial database design ensures that the abstracted reality is modeled as appropriately as possible in a database to meet the user needs. This is achieved by employing database engineering concepts like consistent spatial data models. Modeling really perfectly is not trivial because of the constantly changing complex reality [9]. This is further complicated by diverse user descriptions of how they wish the database to appear.

Database as object design principle

'Turning the database into a consistently behaving object' is the design philosophy adapted with regard to spatial database consistency.

Encapsulation principle

It is geared towards preserving consistency by protecting objects by the methods. Therefore, the users do not have direct access to the database. The goal is to ensure that the database is encapsulated by functions to keep it consistent, ensuring that all the functions adhere to the ACID (Atomicity, Consistency, Isolation and Durability) properties [17].

2.7.2 Server-side functions and triggers programming

In our research project, we use PL/pgSQL procedural programming language that is built in the PostgreSQL database to programming integrity-preserving functions and trigger functions. These functions are programmed to host the web application business rules for validating data submitted. PL/pgSQL supports SQL statements, control structures of PL and parametric functions [9].

Parametric transaction functions

Parametric functions support declared arguments. In our research project, we use data to be passed into the database as arguments. These arguments form the initial stage to enforce domain constraints because data types are declared

at the argument level. In our scenario we use PL/pgSQL to create integrity-preserving transaction functions to validate data consistency at attribute, object, table and database levels. Once the data goes successfully through these stages the submitted data is considered valid and there will be a resultant change in database status [9].

Trigger functions

A trigger is a command specified for the database to automatically execute a defined function when a certain type of event occurs. Triggers are defined to fire when there is UPDATE, INSERT,or DELETE operation. Triggers can be defined to execute either before and after an event, however, after declaration is commonly used [9]. Moreover, UPDATE triggers can be fired only if certain columns are stated in the SET clause of the UPDATE statement.

2.7.3 How constraints are defined in a function

In our research project, we adapted the model of granularity in enforcing the constraints on the bottom-up approach. Therefore, we begin by defining object and tuple constraints in the integrity-preserving functions. The object consistency encompasses the attribute and tuple constraint that are implicitly defined (inherent in the database) and explicitly defined check constraints. All the object and the tuple constraints are be defined in nested IF-THEN-ELSE statements to ensure they are executed as a unit of bottom granularity consistency.

The second level will be to program the table constraints that are executed after attribute and object constraints have been validated. The function nests all the table constraints which consist of complex business rules. Finally, the database constraints are encoded in the integrity-preserving function defined as referential keys and business logic. All these levels constraints in a function are programmed to execute following granularity order, that is, from bottom-up (i.e. fine to coarse level) approach. Therefore, the kernel of the research project is to conceptualize how to implement all the above constraints within PostgreSQL and PostGIS-enabled spatial database and integrated with Django built web application.

2.7.4 Database transaction

It is an array of actions on a database that models the real world and transforms a consistent database from one state to another [17]. If abstracted data violates the consistency defined in the integrity-preserving function, the transaction is rolled back. A database can either execute a transaction fully or not at all, this is called atomicity [17]. Once executed the transaction is durable. When multiple applications are supported by a database, this implies they can execute safely simultaneous transactions concurrently without interference, though in isolation. The preconditions and post conditions to be fulfilled before a transaction is valid are held in constraints. Therefore, predefined constraints determine the transformation of the database from one state to another [9].

2.7.5 Legacy databases

Legacy databases are old databases together with records that were used by previous systems and are available for migration and inheritance into currently developed applications [32]. Legacy databases may have been developed using old database technology [32]. However, it is not good practice to manage two applications running two databases concurrently. Therefore, migration of legacy databases has to be done to have a single database. Databases are designed to be durable because of the value of the data they archive. For this to be achieved and to be used in future, there is need to ensure that all consistency constraints are maintained in the database. Some web application frameworks like Django support legacy databases by mapping and autogenerating the database schema into models. However, not everything is mapped and hence manual editing and cleaning has to be done to avoid making errors during data modification in the database [12].

2.8 Conclusion

A number of web applications were studied but not all could be used to implement and evaluate spatial database consistency in web applications. Django, Turbo Gears and Pylons have different programming capabilities for different types of application based on the user needs [8]. Therefore, we settled on Django because of some strengths it possesses with regard to handling spatial data.

2.8.1 Strengths of the choice of Django

Django, a python-based web application is chosen as a testing platform for implementing the granularity based consistency constraints as well as a design method. Django is free software that allows for further extension to enable the developers come up with a suitable tool to meet user needs [12]. Django also works well with a number of open-source spatial databases like PostgreSQL [8, 12]. Moreover, Django has an extension package called GeoDjango that enables creation of web GIS applications [18]. The GeoDjango complies with OGC simple feature data types. GeoDjango also works with other spatial web visualization and rendering tools like OpenLayers and Google Maps. In addition, Django has a built-in admin interface that handles data manipulation like CRUD, browsing and search [8, 12, 3].

Chapter 3

Spatial consistency design for Web 2.0 applications

3.1 Introduction

In this chapter, we discuss the design aspects that help solve spatial database consistency problems in web application frameworks. In the previous chapter on literature review, we delved into web frameworks and identified problems affecting web frameworks with regard to maintenance of spatial database consistency. We looked at how feasible it is to have the integrity controls within the database, that is, how to bring about sharing of a database by multiple applications without compromising the integrity of stored data. Therefore, we planned to have integrity-preserving functions that handle constraints at various taxonomic granularity.

In this chapter, we focus on requirements analysis and system design. We design how integrity-preserving functions can cater for taxonomic granularity during enforcement of integrity constraints. We also design how to extend the Django web framework to accommodate Remote Procedure Call API calling capabilities. Therefore, our objective is to have a design system prototype for development that extends and improves handling of spatial database consistency in the Django web application framework.

For our research project, we employ a waterfall design model for software development as shown in Figure 3.1.

3.2 Requirement analysis

The main goal of this research project is to come up with a design method that will ensure spatial database consistency in web applications. The need to have successful spatial consistency will bring Django architecture and constraints enforcement mechanism into the center stage of detailed requirements analysis. We look deeply into Django and PostgreSQL system architecture to decipher the system requirements needed to meet the user requirements to have spatially consistent databases. During analysis, notes are made of the existing modules to tweak, decision points, and transactions handled by the present

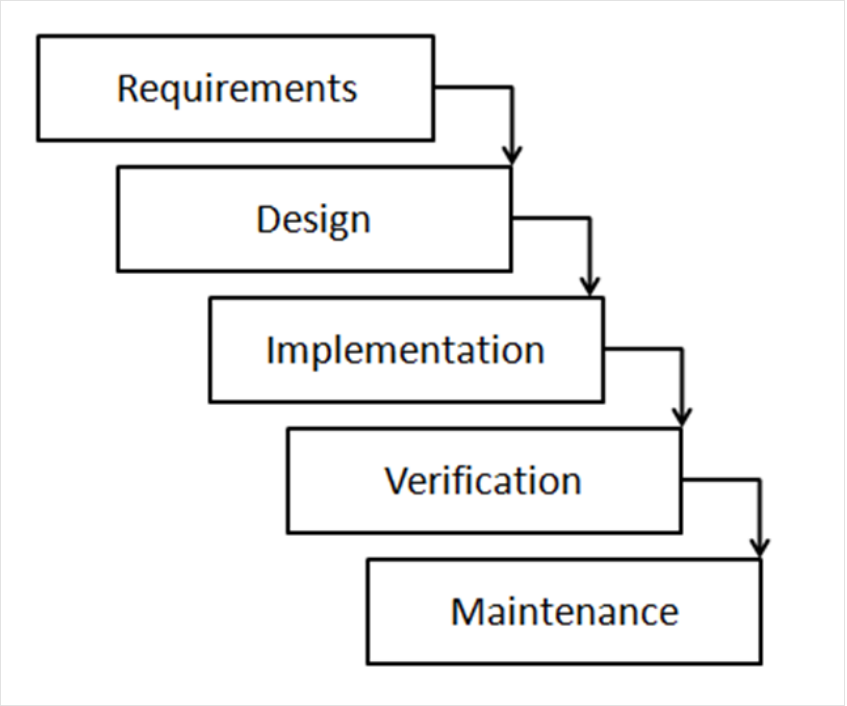

Figure 3.1: Waterfall design model (30)

system.

The user requirements must be declared clearly to avoid any ambiguity when designing a system to address the user needs. Any fuzzy description should be clarified to avoid misrepresentation. Requirements are generally split into two main portions, that is, functional requirements and non-functional requirements [10].

3.2.1 Functional requirements

Functional requirements constitute the functions of a software system or its components. Functional requirements consist of input, behavior and output as shown in Figure 3.2. These functional requirements include data manipulation, transfer, validation or processing functions. The functional requirements are defined in the system design and can be depicted using use case diagrams in UML.

Functional requirements require us to study deeply the Django and GeoDjango operations and to understand their capabilities in handling spatial processing tasks. We adhere to the rules of web design such that we did not violate the MVC architectural design. This ensures that we end up with an improved system that still has Django built-in benefits as well as the added functionality. Moreover, our goal is to have minimum code, hence, we spend time in studying the framework to ensure that we call and reuse existing classes and methods.

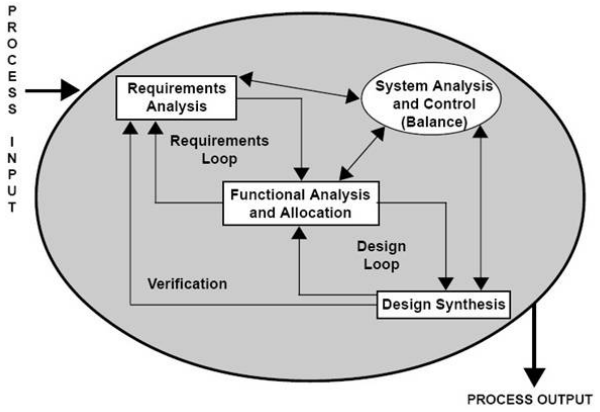

Figure 3.2: Functional requirements diagram (10)

3.2.2 Non-functional requirements

Non-functional requirements impose quality aspects to the implementation. It can not be stated explicitly within the system design. In our research project, we looked into the following non-functional requirements:

- Spatial database consistency demands that the system adheres to the prescribed to the integrity controls based on taxonomic granularity levels.
- The constraints should reside in the database back-end.
- The database must be reusable by other web applications.
- The application must comply with the DRY principle and should be loosely coupled.
- As a scalable system, it should accommodate increased volume of data without requiring the system to be redesigned.

3.3 Constraints design method to promote spatial database consistency

When developing spatial consistency integrity controls for web application, the taxonomic granularity order will be adhered to. The constraints enforcement in the database are based on four taxonomic constraints granularity levels namely: attribute ,object, table and database constraints as explained in Section 2.5. When data that abstracts reality passes the validation test in a spatially consistent database, it is considered to represent the intended reality and therefore it is committed changing the database status.

Just like the design of system components, the creation of constraints have to comply with a certain design method. This is due to the fact that various constraints that validate data operate differently and are supported differently.

For instance, referential keys and check constraints cannot be similarly defined. This is because a check constraint hosts data it uses to validate input data within the conditional statement used for validation. That is, it does not refer within the modified relation or from other relations. This because users can modify the data referenced resulting in inconsistencies. On other hand, referential key constraint depend on other tables to check validity of the data submitted. However, referential key constraints and check constraints share the similarity of being executed based on trigger mechanism.

3.3.1 Constraint elements

Some of the elements that constitute the basic components and operation of a constraint are discussed below.

Nature of input data to be validated

The data to be validated by a constraint can be supplied either as parameters of an integrity-preserving function executing the constraint or stored temporarily as a record, table or database where they are called by validation expression. Validated data can either be attribute or spatial or both types.

Nature of reference data used by a conditional statement for validation

Data used for validation can be explicitly defined in a conditional statement like in check constraints. The condition statement can also reference data from other relations or from within the table to be modified.

The validation conditional statement

Conditional statements that return boolean value are used to implement constraints used to validate the supplied datasets. If a condition is fulfilled, then the condition statement returns true. If a condition is not fulfilled, then the condition statement returns false.

Action after a conditional statement is executed

If a conditional statement returns a boolean of type TRUE.

- The data is committed into a database resulting in change of the database status.
- Or, the data is processed before updated into a database.

If the conditional statement returns a boolean of type FALSE, then two of the actions can be defined by a developer:

- A warning message to be sent to the user to correct the data.
- Or, the data can be rectified using complex pre-processing functions. For instance, if a river is meant not go beyond a given area of jurisdiction, then the unwanted overshoots of the river can be sliced out.

3.3.2 How constraints are implemented

Once constraints have been created, they have to be utilized, that is, used to validate data. Validation can be done before a database update , or after an database update, or partly before an update and partly after an update. The three validation techniques are discuss below:

Validation done before conducting an update

In this approach, created constraints validate data before an update transaction function is executed. To achieve the before update validation technique, integrity-preserving functions are used. Validation can be done at the database layer or the application layer. We found out that functions defined at a given layer, that is, either database or application, maintained consistency in their respective layers. However, based on the advantages of each layer as discussed in Section2.6.3, database validation functions can be called into web application for better performance.

Validation done after conducting an update

In this approach, an update transaction function is executed before validation is done. This technique is made possible by the use of trigger mechanism that fire when an update is made. Mostly, triggers are used in databases. However, triggers implementation compared to integrity-preserving functions implementation have advantages and disadvantages as discussed in Sections 3.3.12and 3.3.13.

Validation done partially before update and partially after update

In this approach, validation process is done partly before and partly after an update transaction. Most of the web applications like Django use this approach. Django conducts validation using explicit constraints at the application layer and then uses trigger based inherent constraints at the database layer.

The limitations of application layer validation is the lack of central location to manage the constraints and limited support for spatial complex business logic. This prompted us to conceptualize ways to improve spatial database consistency in web application frameworks. We proposed that, if web applications can utilized database functions whether created from a database or from a web framework using an SQL wrapper then spatial consistency will be improved. We also proposed that, if validation constraints are created and implemented based on constraint granularity level then database spatial database consistency is improved because they are implemented in an orderly manner and can reused.

3.3.3 Design of constraints as database integrity-preserving function (ψ)

Algorithm 1 shows the general design method employed when creating database integrity-preserving function of various constraints levels. The elements discussed in Sections 3.3.1 and Section 3.3.2 form part of the of constraints design requirements. Based on the business rules supplied constraints can perform simple or complex conditions. Since the functions reside in the database, then all the constraints will inherently be stored in the database.

Algorithm 1: *General design of integrity-preserving functions (ψ)*

Require: Create an integrity-preserving function ψ for a given constraint level.
Require: Data N to be validated is supplied as parameters of the function (arg1 datatype,.. argn datatype).
Require: Store referential validation data V in a temporary storage (as record or table).
Require: Database status D.
Ensure: Validation if of members either attribute, object, table or database.
1: **if** $Condition\ considering\ value$ (V) $is\ FALSE$ **then**
2: Abort validation, Raise warning to the user, ψ(N) does not hold.
3: **else**
4: **return** TRUE, hence data N is valid, $\psi(N)$ holds.
5: **end if**
If all the constraints at each the four integrity levels are met within functions, then $\psi(N)$,$\psi'(N)$,$\psi''(N)$,$\psi'''(N)$ hold. Therefore, an update transaction function $\mu(i)$ updates the database changing its status, $D \leftarrow D'$.

3.3.4 Design of integrity-preserving function at each constraint granularity level

The design method discussed in Algorithm 1 is further made specific for each of the four constraints levels as discussed below, that is, attribute, object, table and database as shown in Algorithms 2, 3, 4, 5 respectively. According to our design method, an integrity-preserving function contains constraints for a given single granularity constraint level.

3.3.5 Attribute integrity-preserving function (α)

An attribute integrity-preserving function validate data that is of attribute record level. The Algorithm 2. For instance, data J of supplied as arg1 parameter of the function is validated by a condition statement that uses validation value C. If the condition is not fulfilled the function aborts and sends warning to the GUI. If the conditions is fulfilled the function returns True.

Algorithm 2: *Design method of attribute integrity-preserving function (α*

Require: Create a function α that has parameters (arg1 datatype, arg2 datatype, ...argn datatype).
Require: Data to be validated J is contained in function arguments.
Require: Data used for validation C contained in the conditional expression.
Ensure: Validation should be within data structure of type $attribute$.
1: **if** $Condition(C)$ $checking$ $(J.arg1)$ is $False$ **then**
2: Abort, and send warning to the GUI.
3: **else**
4: **return true**,
5: **end if**
6: The function $\alpha(J)$ holds when data J is valid.

3.3.6 Object integrity-preserving function (β)

Object integrity-preserving function are created as shown in Algorithm 3. The tuple members X and K are validated based on a given condition. If the condition is not fulfilled the function aborts and sends warning to the GUI. If the conditions is fulfilled the function returns True.

Algorithm 3: *Design method of object integrity-preserving function (β)*

Require: Create a function β that has parameters (arg1 datatype, arg2 datatype, ... argn datatype).
Require: Data to be validated X and K is contained in function arguments.
Ensure: Validation is amongst members of an object.
1: **if** $(X.arg3)$ $Condition$ $(K.arg5)$ is $False$ **then**
2: Abort, and send warning to the GUI.
3: **else**
4: **return true**
5: **end if**
6: The function $\beta(X, K)$ holds, then data X,K are valid.

3.3.7 Table integrity-preserving function (γ)

Algorithm 4 illustrates how integrity-preserving function is created. Then, it searches within the table members and checks using threshold R, for instance, whether the table level constraints is fulfilled before the entire data supplied can be valid for a transaction. If the condition is not fulfilled the function aborts and sends warning to the user. If the conditions is fulfilled the function returns True.

3.3.8 Database integrity-preserving function (δ)

Algorithm 5 shows how database integrity-preserving function is created. First, the function is created with parameters that have their data types included.

Algorithm 4: *Design method of table integrity-preserving function (γ)*

Require: Create a function γ that has parameters (arg1 datatype, arg2 datatype, ... argn datatype).
Require: Check value R can be used as validation data threshold stored in condition expression.
Ensure: Check within members of a table, before all function parameters are valid.
1: **if** $(Condition(count\ members)\ <\ R)\ is\ False$ **then**
2: Abort, and send warning to the GUI.
3: **else**
4: **return true**
5: **end if**
6: The function $\gamma(all\ parameters)$ holds, then all data supplied function parameters are valid.

Then, data L used to in the condition to validate data U is stored in temporarily within the function. Then, Validation is between members across tables. If the condition is not fulfilled the function aborts and sends warning to the user. If the conditions is fulfilled the function returns True.

Algorithm 5: *Design method of database integrity-preserving function (δ)*

Require: Create a function δ that has parameters (arg1 datatype, arg2 datatype, ... argn datatype).
Require: Data to be validated U is contained in function arguments.
Require: Data used for validation L is stored in temporary storage within the function.
Require: L $\leftarrow (SELECT\ *\ FROM\ table\ B)$ if data is gotten from $tableB$.
Ensure: Validation should be between across tables members.
1: **if** $Condition(C.Column1)\ checking\ (U.arg4)\ is\ False$ **then**
2: Abort, and send warning to the GUI.
3: **else**
4: **return true**
5: **end if**
6: The function $\beta(U)$ holds, then data U is valid.

3.3.9 Design method of update transaction function (λ)

Algorithm 6 illustrates the method developed to create transaction update functions when the all integrity-preserving functions return True value.

3.3.10 General design of an integrity-preserving trigger function

An integrity-preserving trigger function for each constraint level are created almost in the same way as shown Algorithms 2, 3, 4, 5 but with two major

Algorithm 6: *Create update function (λ)*

Require: Create a update function λ that has parameters$(arg1, arg2)$.
Require: Data to be updated $arg1$ and $arg2$.
Require: Database status H.
Ensure: Call all the integrity functions α, β, γ, δ and perform them within from update function.
1: **if** $(SELECT\ Integrity\_functions$ (arg1,arg2) $= True)$ **then**
2: Update genus table.
3: Database state changes $H \leftarrow H'$.
4: **end if**
5: The function $\lambda(H)$ holds when data H is valid.

differences. First, the function is created as a trigger function. Secondly, the triggering condition is set to fire if there is a detection of any new update that intends to change status of the database. If there is no update, the triggers remain dormant. Suppose integrity-preserving trigger functions α', β', γ', and δ' represent constraint level attribute, object, table and database respectively. The trigger functions $\alpha'(W)$ and $\beta'(W)$ and $\gamma'(W)$ and $\delta'(W)$ hold for data W validated, then database state is changed. Algorithm 7 shows the design technique of the trigger functions.

Algorithm 7: *General design of an integrity-preserving trigger function*

Require: Create an function α', β', γ' and δ' as trigger function that return true.
Require: In each trigger function α', β', γ' and δ' ,insert a $trigger\ statement$ that fires after a new update event.
Require: The data to be validated is W.
Require: The database state is Z.
1: **if** $data\ W\ is\ not\ updated$ **then**
2: Trigger remain dormant.
3: **else**
4: Fire trigger and to validate W.
5: **end if**
6: **if** $\alpha'(W)$ and $\beta'(W)$ and $\gamma'(W)$ and $\delta'(W)$ do not hold **then**
7: Abort, and send warning for the indication the constraint level violated.
8: **else**
9: Commit transaction and change the database status $Z \leftarrow Z'$.
10: **else**
11:
12:
13: **end if**

3.3.11 Mechanism of executing integrity-preserving functions

In our design method, we explain two approaches employed when executing the integrity-preserving functions in order to conduct data validation processes. These approaches are: use of triggers functions and Remote Procedure Call API defined in the application layer. The two approaches comply with the philosophy of having the constraints reside in the database.

3.3.12 Execution of integrity-preserving functions using triggers

Triggers are fired when there is an UPDATE, DELETE and INSERT event. The triggers can be set to fire before or after an event. The Algorithm 8 shows how the integrity-preserving triggers functions for each constraint level are implemented in a database. To begin with, in a transaction, the entire input datasets are updated using a transaction function κ resulting in a new (B') database state. The a triggers for each constraint level (i.e. attribute,object, table and database) is fire to test the validity of the new database state and returns boolean. If data violates any constraint in a given integrity-preserving trigger function, then the transaction is aborted. However, if all the constraints in each integrity-preserving trigger functions is fulfilled, then the transaction is committed.

Algorithm 8: *Execution of integrity-preserving functions using triggers*

Require: Create integrity-preserving trigger function γ,θ, π, ω for each of the four constraints levels

Require: Update data q to be validated into a database of state B, resulting in new state B' using traction function κ.

1: **if** $\gamma(q)$ and $\theta(q)$ and $\pi(q)$ and $\omega(q)$ do not hold **then**
2: Abort transaction, send warning to user indicating the constraint violate.
3: **else**
4: Commit transaction, $B \leftarrow B'$.
5: **end if**

The integrity-preserving trigger functions are used to implement validation and modification of data in a database, they reside in a database guaranteeing the integrity of a database. However, it has a number of limitations. For instance, the constraints are opaque to the client application that has access and user rights to modify a database. Moreover, when complex business logic is used to validate large datasets, especially spatial data, the validation process takes a long time. Since multiple triggers can fire per event, trigger firing rate also increases with increase in the number of users updating a database, hence the validation process is further slowed. Also the cascading effect of triggers can slow the processing of data, that is, if other triggers are fired.

3.3.13 Implementing integrity-preserving functions using a Remote Procedure Call API

Implementation using Remote Procedure Call API aims at addressing the limitations resulting from the use of triggers as discussed in Section 3.3.12. The Algorithm 9 shows how an update function λ is called into an application by Remote Procedure Call API. The update function λ then call all the integrity-preserving functions in an orderly manner, starting with the attribute α, then object function β, then table γ, and finally the database δ integrity preserving function. If all the integrity-preserving functions return True, then the update function λ (as designed in Section 3.3.9) is used to commit the transaction resulting in change of the database status. If False is returned, then a warning message is sent to the GUI.

Algorithm 9: *Implementing integrity-preserving functions using a Remote Procedure Call API*

Require: Create a Remote Procedure Call API.
Require: Data g to be validated.
Require: Integrity preserving functions α, β, γ, δ.
Require: API calls update functionλ.
Require: Update function λ calls all integrity functions. It then prompts the integrity functions to perform the below test. If all return True, then modify database status K.

if $\alpha(g)$ does not hold **then**
    Abort, Raise warning.
else
    return true
end if
if $\beta(g)$ does not hold **then**
    Abort, Raise warning.
else
    return true
end if
if $\gamma(g)$ does not hold **then**
    Abort, Raise warning.
else
    return true
end if
if $\delta(g)$ does not hold **then**
    Abort, Raise warning.
else
    return true
end if
if $\alpha(g)$ and $\beta(g)$ and $\gamma(g)$ and $\delta(g)$ holds **then**
    Database state changes $K \leftarrow K'$
end if

The Remote Procedure Call API implementation has a number of advan-

tages over the trigger implementation. To begin, constraints are made known to the application while in triggers they are opaque, therefore the triggers running behind the scene can be forgotten and a times difficult to debug when errors arise. Data validation process is faster because processing is done one at time following the granularity constraint levels unlike in trigger implementation where all the triggers fire sporadically upon updating the database.

3.3.14 Summary of steps followed when implementing constraint design method

Constraint design method prescribes a four-step method to promote spatial database consistency in web applications. The method was developed after taking into consideration constraints design parameters and techniques discussed in Section 3.3.

1. Understand both simple and complex business logic, then define each basic constraint based constraint granularity levels (i.e. attribute, object, table and database).

2. Then an integrity-preserving function is created that wraps each constraint in a function based on granularity levels. The integrity function return boolean.

3. Then create an update transaction function that calls the integrity-preserving functions. If the called functions return True then update statement commits the transaction into a database.

4. To utilize the transaction update function, we use a Remote Procedure Call API.

It should be noted that if the integrity-preserving functions are implemented using triggers, then the fourth step is avoided. However, because of the limitations of triggers being slow and opaque, we use a Remote function call Remote Procedure Call API.

3.4 Design of Django to support Remote Procedure Call API

In this section we discuss the redesigning of Django to support the creation of Remote Procedure Call API that will enable web applications to call integrity-preserving database functions to be used for validation and hence promote spatial database consistency. We begin by discussing the system components of Django and GeoDjango that need to be taken into account in order to create a Remote Procedure Call API that will enable the constraints to reside in the database back-end. Then we will compare the current Django design with the new design with respect to handling validation constraints. Then we summarize with a new layout of the new prototype design.

3.4.1 Functional requirements use case diagram

Key functional requirements to be contained in the new system are depicted using UML use case diagrams as show in Figure 3.3. A use case simplifies the specifications and avoids the use of jargon making it easier to understand. In our scenario, a use case diagram is used to model internal working of the system that needs to be extended and those that participate in extending it. Therefore, the entire system is not captured in the use case and we only focus on those major components that need to be modified. The use case does not capture the non-functional requirements like reusability. Ultimately, the modeling of the system activity at the requirement analysis phase, helps to avoid errors that may result during the system implementation, that are harder to correct at a later stage. This guarantees success of the project. The Figure 3.3 shows how the system's actors interact with system functions in use cases.

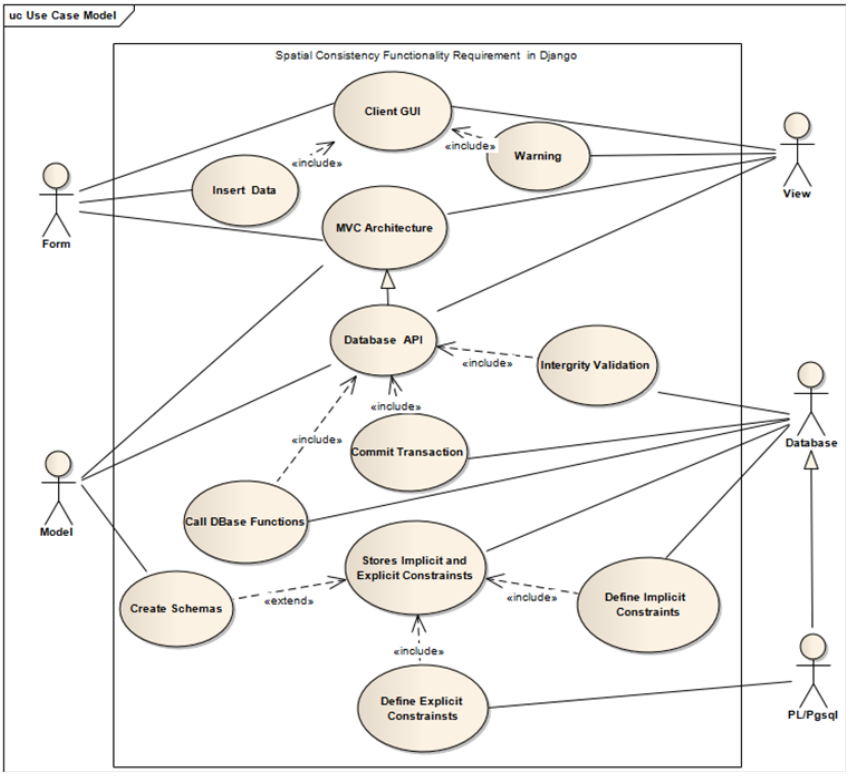

Figure 3.3: Functional requirement use case diagram

The functional requirements use case diagram in Figure 3.3 represents the system improvement requirements needed to archive a spatial database consistent web application. The Django Template (i.e. Form or Viewer) constitutes part of the graphical user interface. Within the GUI exist form fields for data entry. In addition, the GUI displays output feedback messages sent to notify the user whether the data was submitted successfully or rejected. The Model, Template and View (MTV) constitutes part of the MVC architecture. That is, the central engineering concept of web applications should be maintained together

with all its functions.

PostgreSQL and its PostGIS extension are chosen as the spatial database of choice for the system. A Remote Procedure Call API needs to communicate with the model and view when it calls functions and commits transactions. On the other hand, the database serves the purpose of storing the functions, validating data and executing transactions. The model uses its ORM to create the schema, though an option is provided for the database to use raw SQL to create schema. The implicit and explicit constraints are meant to reside in the database. Implicit constraints are defined inherently within the database while the complex business logic is defined using PL/pgSQL server-side procedural language and stored in the database as discussed in Section 3.3.3.

3.4.2 Design method used to create a Remote Procedure Call API in Django

The main goal of a Remote Procedure Call API is to ensure that a web application can call integrity-preserving functions, use the functions to validate data and be able to convey warning message back to the user.

Design of Remote Procedure Call API component in model.py

The component of API that resides in model module is charged with responsibility of providing communication with the database, requesting for database transactions and acting on feedback from the databse. Algorithm 10 explains how the API component in model.py is created.

Algorithm 10: *Design of Remote Procedure Call API component in model.py*

Ensure: Django application modules model.py is opened.
Require: Call database connection and transaction classes into model.py.
1: In model.py create a class with an appropriate name (e.g. Dbase_funct_call).
2: Within the class create a method (e.g. func_trasact).
3: The func_trasact method takes parameters, that is, data received from view for validation.
4: Create a variable (e.g. genus) to assign cursor method that takes data to be validated (i.e. method parameters) and the name of integrity-preserving function as its arguments.
5: Define execute cursor to ask the integrity function to evaluate the data.
6: **if** $Validation\ process\ is\ complete$ **then**
7: Send to the view the message returned by the database validation function.
8: **else**
9: Wait for response from database.
10: **end if**
Ensure: Method returns response (i.e. success or message).

Design of Remote Procedure Call API component that reside in view.py

The component API in view module is responsible for sending and receiving information to and from Form/GUI. It utilizes the methods defined in the API class in the model module. Algorithm 11 explains how the API component in view is created.

Algorithm 11: *How to create Remote Procedure Call API component in view.py*

Ensure: Django application modules view.py is opened.
Require: Call Dbase_funct_call class created in model.py.
Require: Call Django Form class {this enable the API to receive data from GUI and send warning messages to GUI}.
1: Create a function in model module (e.g. called render_post).
2: Create a variable that uses request method from Form to establish connection with the form field in GUI.
3: The render_post function inherits the func_trasact method from Dbase_funct_call class. Within the class create a method (e.g. func_trasact).
4: Create a variable (genus) that call method (func_trasact) in class Dbase_funct_call that takes values received from the GUI as its arguments.
5: **if** $Data\ is\ received\ from\ GUI$ **then**
6: Send data to API component in model.py module.
7: **else**
8: Send message received from the database via model API component to GUI.
9: **end if**
Ensure: The function returns render response (i.e. present information for display by url.py).

3.5 PL/pgSQL wrapper design

The PL/pgSQL wrapper once created will provide will enable Django to define PL/pgSQL functions and also triggers that reside in the database using Django functions. Therefore, the need of running raw SQL codes in Django will be avoided. With the wrapper in place, we will simplify the creation of functions that reside centrally in the database. The PL/pgSQL wrapper that will extend Django ORM should be able to:

- Enable a developer to create and delete both functions and trigger functions.
- Allow the declaration of data types for function arguments.
- Allow unlimited entries of parameters.
- Return either boolean, integer, void , real or varying character.
- Allow one to pass a declare variables and also create temporary records within the function.

- Strive to include all the built-in PL/pgSQL statements like UPDATE, BODY , END, RETURN, SELECT FROM WHERE , REFCURSOR, INSERT , DELELE ,OLD, NEW, IF-THEN-ELSE, LOOP, TG_OP, TG_NAME among others.

- Call and reuse other built-in database functions.

- Though not a must, the PL/pgSQL wrapper should also be able to translate an existing PL/pgSQL function back into a Django method.

3.5.1 How PL/pgSQL wrapper is created

Concatenation of built-in statements and input variables then returning a string variable, is the fundamental principle behind the design and creation of a PL/pgSQL wrapper as explained in Algorithm 12.

Algorithm 12: *How PL/pgSQL wrapper is created*

Ensure: Create a class (e.g. plpgsql_wrapper) that has a number of methods.
Require: Create methods that each takes argument variables.
Ensure: Every method created should return a string value.
Ensure: Every method concatenate built-in SQL statements with variables passed to them.
Ensure: Between concatenated statements, a space should inserted to avoid creating a continuous string that would raise errors.
Ensure: __init__.py file is created that will trigger the creation of 'CREATE FUNCTION' statement and closing statement 'LANGUAGE 'plpgsql" when the a create class is called. Then all the concatenated strings will be inserted inside at the end, hence avoid omission errors.

3.5.2 How PL/pgSQL wrapper is used to create database functions

The PL/pgSQL wrapper enables a developer to create Django functions that generate PL/pgSQL codes used to create database functions. A Django function works on the principle of inheriting and passing variables to methods of class wrapper. The wrapper concatenates and returns string variable. The Django function executes the generated code by as raw SQL statement. A Django function that generate PL/pgSQL code can either generic or specific to certain business rules. For instance, for simple validations, Django functions can be reused to create database functions by varying the input parameters, but for complex validation rules, a developer has to create user-defined functions.

3.6 Comparison in terms of data flow between the Django design and proposed design

The proposed design method has an impact on data flow, that is , starting from data entry point at the GUI, then going through validation checks at application and database layer and finally committed to the database for storage. Therefore, we illustrate the difference in two designs by discussing data flow in each design method.

3.6.1 Current design data flow in Django web framework

Figure 3.5 shows how data flows currently when a Django web framework is used to build a web application. Take note, that Django follows the MCV architecture but interchanges its layers in terms of terminology. The View in Django is considered to be the MCV architecture's controller, the Template is considered as the View in MVC architecture's View and the model remains the same, hence Django's MTV architectural design as shown in Figure 3.4.

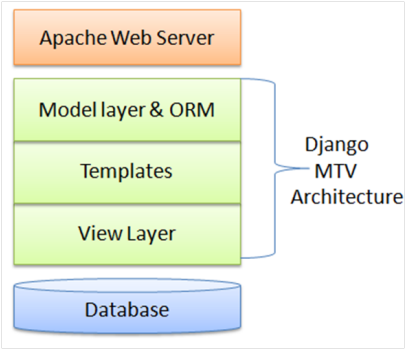

Figure 3.4: Django's MTV architecture

Bear in mind that Django together with GeoDjango has been used to create a web application that is connected to a spatial database ready for data entry. According to the current design method, the flow of data in the deployed spatial web application is illustrated in Figure 3.5. First, the user enters the data into the GUI form fields (Template) and presses the submit button. Then, the Django view layer that has similar roles as the controller, is responsible for getting data in and out of the database while it communicates with model and ORM. The view hosts the web application complex business logic that is responsible for consistency checks.

Therefore, the application level validation first takes place in the view. If an error occurs, an error message is send back to the GUI client. The user is then expected to cross check and enter the correct data. However, if the data is found to comply with defined business rules, the view sends the data into the database where the inherent constraints further validate the data. If the data is found to be correct the transaction is submitted into the database for storage. Otherwise, if there is violation of these constraints, an error message is send

back to the client user interface. The same process is repeated again for the next transaction.

Based on this design the spatial data consistency is compromised because constraints are not maintained in a central location . This is because Django consistency controls reside partly in the application layer and in the database layer. Complex business rules cannot currently be defined in the database by Django. Therefore, Django define and maintain complex business rules in the view. The Django together with GeoDjango supports the creation of limited spatial business rules in the view. Forms support validation of simple rules like data types.

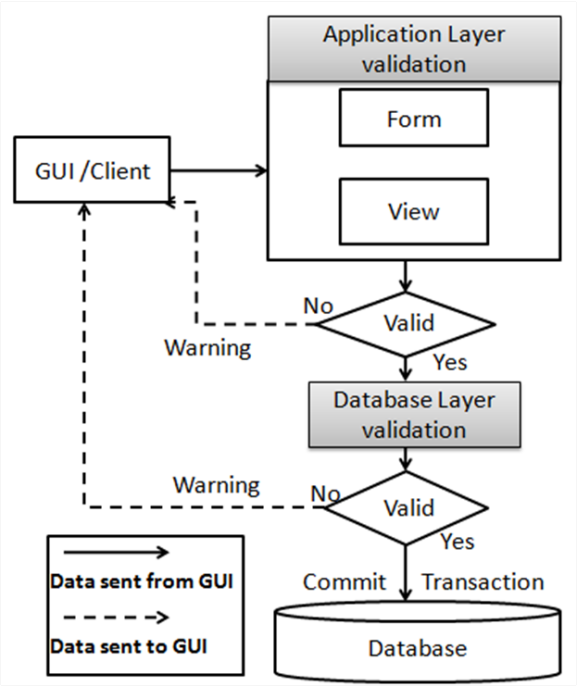

Figure 3.5: Django design data flow

3.6.2 New design data flow in extended Django web framework

Considering the existing Django design, there are limitations when it comes to guaranteeing spatial database consistency. Therefore, the new extended design proposes the creation and maintenance of all the integrity constraints within the database. According to the new design, the expected data flow is shown in Figure 3.6.

First, the user enters the data into the GUI form fields (Template) and presses the submit button as shown in Figure 3.6. Then, the Django View layer communicates with model and ORM and allows movement the data in an out of the database via a Remote Procedure Call as discussed in Section 3.4.23. However, the view in the new design is not used to store the complex spatial business logic, hence, the validation stage is left out at the application level.

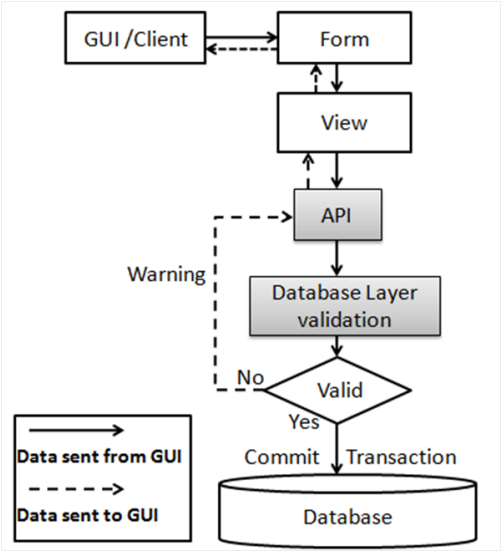

Figure 3.6: New design data flow

Therefore, the spatial business logic and all the integrity controls are stored in the database layer.

The API is designed to call integrity-preserving functions from the spatial PostgreSQL database. Moreover, the API enables data to be wrapped in predefined database functions, which then forwards the data through the database validation based on taxonomic granularity before committing it into the database for storage. Once the data reaches the database, it goes through validation process. If the dataset is erratic an error message is send back to the GUI of the client browser. The user is then expected to cross check and enter the correct data. However, if the data is found to comply with defined business rules the API commits the transaction and stores data, hence, changing the status of the database. The same procedure is observed again for the next transactions.

In this approach we get rid of redundancy of double validations at the application and the database levels. Moreover, we achieve the goal of having the integrity controls reside in the database layer. This will promote the consistency of spatial web application because of the earlier reasons stated in the literature review like ability of the database to be access by multiple applications without messing the integrity of the database.

3.6.3 Prototype design architecture

After taking into consideration the system requirements analysis and delving deep into the operation mechanism of Django web application framework, a new design architecture for a prototype web application framework that extends Django was developed. Figure 3.7 illustrates the new stet-up of extended Django with an API that promotes spatial database consistency.

The design give provide options that improve the building of spatially consistent web applications. For instance, the classical database design can be

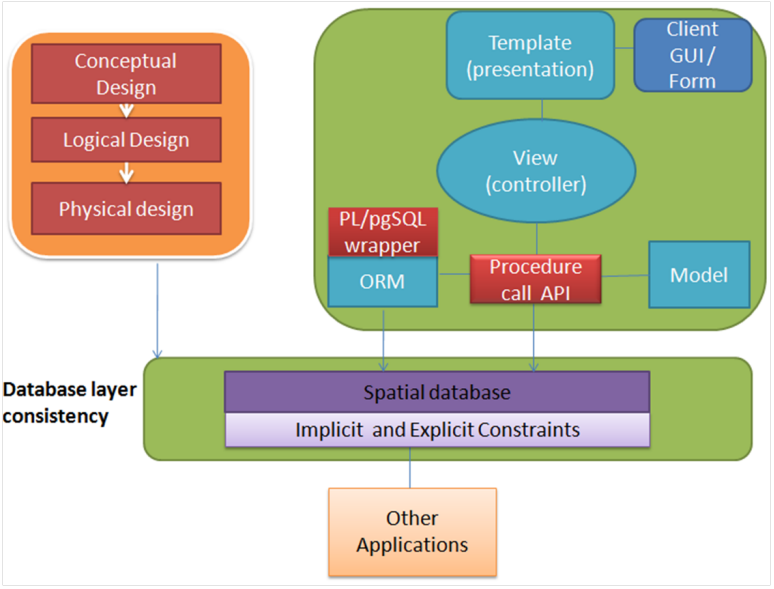

Figure 3.7: Prototype design architecture

used to create schema and later the model and the ORM notified of the built-in schema and the inherent database constraints. A modeling language like UML is used model the complex relationship between the schemas and to define any multiplicities. Therefore, the ORM can be used at liberty to create schemas or not, based on the developer's desire and choice.

A Database API is another addition into the design. It is mainly used to call built-in PostGIS functions and transact with the database. PL/pgSQL wrapper is also proposed to extend Django to support the creation of database functions. Other applications and clients can as well plug into the database and do some edits so long as they do not violate the consistency rules.

3.6.4 System infrastructure architecture

The system architecture is divided into client, web server and database as shown in Figure 3.8. It constitutes software, hardware and network.

Client

The user interacts with the web application via an Internet browser. This enables many users across the world to have access to stored information or contribute their data via a GUI form into the database. The hardware used at the client side include desktops, PDA, laptops or Mobile phones.

Web server

The server is charged with the responsibility of responding to HTTP requests made by distributed clients via the Internet. Apache web server which is em-

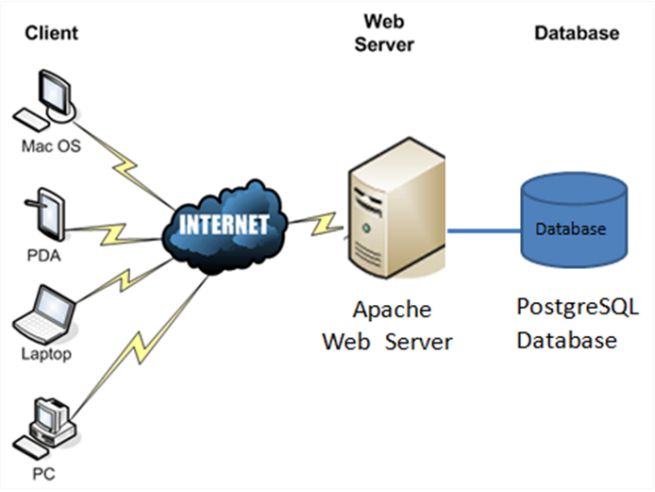

Figure 3.8: System architecture

ployed in this research project is also used to receive content from the client. For instance, uploaded files or submitted web data entries.

Database

We use PostgreSQL database which is an object-relational database management system (ORDBMS) and its spatial extension called PostGIS for this research project. Database models together with business logic are created and stored within PostgreSQL and PostGIS. It supports the SQL statements and has a server-side procedural programing language called PL/pgSQL.

3.7 Introducing pre-processing of data before storage in databases

The discussed design method will pave way for smart databases that will not only view databases as repository to store spatial and attribute data but also put predefined meaning into data by enforcing constraints and other business rules. These rules are enforced through the use of integrity functions as discussed in Section 3.3.3. By adhering to these rules defined in the database, meaningful datasets that have been validated and processed accordingly are committed into a databases during a transaction.

This approach is much different from the conventional GIS practice where users could only draw datasets into the spatial database in a GIS environment and save the edits without subjecting the data to any validation checks. Therefore, a lack of meaningful validation or built-in processing functionality that interacts with the entered vector and attribute data before storage in spatial database, will result in overall erroneous datasets despite the good accuracy of the input datasets. This occurs mainly if the entered data is related other to existing base datasets within the database.

3.8 Conclusion

We have explained constraint design method and the system architectural design phases of a spatially consistent database, that is, the framework when extending a Django web framework by creating an API and techniques of creating integrity-preserving functions. The proposed design method, was used to develop a web-based spatial application, that is, be used to test the and evaluate the design's viability in promoting spatially consistent databases. The proposed design methods affects the application development method of web application using the Django web framework as explained in Chapter 4.

Chapter 4

Implementation and Evaluation

4.1 Introduction

In this chapter, we focus on implementation and evaluation of the design method as proposed in Chapter 3. We delved into the programming process using Python programming language to extend the Django web application framework. We also built a web application that maintains spatial database consistency that is based on an Amazonia avian distribution use case. The web application has an additional API for calling database integrity-preserving functions residing in the database. Therefore, we also did server-side database programming to encode the business logic within the database as defined the constraint design method in Section 3.3. The database of choice is PostgreSQL with its POSTGIS extension.

We built the web application taking into consideration the two research project's main philosophies, that is, maintaining constraint validation in the database, and enforcing the validation of constraints based on taxonomic data granularity as postulated in the design method. The ultimate goal is to guarantee spatial database consistency.

The built system is tested using Amazonia avian distribution data to ensure that it performs as designed. Testing is critical during implementation and is primarily geared towards mitigating costly bugs in the system.

4.2 Validation in Django

We begin by showing how Django handles validation of input data at both the application and the database layer. Then, we go through two implementation approaches that we developed to enrich Django according to the design pattern proposed in Chapter 3.

4.2.1 Layout of Django validation process

Figure 4.1 shows the approach of how Django handles validation. The model generates the schemas and inherent database constraints. These inherent constraints are the only constraints that reside in the database layer. The view module is used to enforce the application's complex business logic.

Check constraints are not defined by the Django model. However, the GeoDjango extension supports definition of geographical fields that take SRID and geometry type constraints. The class below in the model module shows that the included polygon geometry is of SRID type 4326.

```
class Distrib(models.Model):
    speciesid = models.CharacterField(length=50)
    name = models.CharacterField(length=50)
    geom = models.PolygonField(SRID: 4326)
```

From our research studies, we learnt that the model is handicapped by the failure of some databases, like MySQL, to support check constraints. However, for a PostgreSQL database it is not an impediment but it limits the creation of complex business rules within the model. This is because the model does not support complex business logic. This is because Django stores the complex business logic in the application layer and not in the database back-end. Therefore, it rules out the creation of functions within the model. Current, the Django ORM SQL wrapper cannot define complex functionality that do complex operations like server-side programming. This is the reason why Django employs the view to define functions that do application layer data validation as shown in Figure 4.1. The form checks that the input data type is valid and clean to be supported by the database. The view is used to create user-defined functions are then passed to the form for data type validation of the result or else a validation error would be raised. Appendix C and D show an example of how form support validation and how form sets are used in a view (which is similar to Controller in the MCV architecture) for validation including complex rules definition.

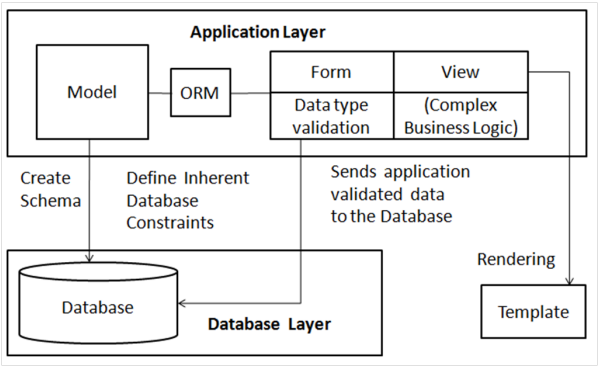

Figure 4.1: Django data validation layers

The weakness of this design implementation approach is that the explicit constraints reside in the application layer. This implies that the philosophy of central location of integrity rules in the database where the data is stored is cannot be achieved. Also, it is hard to monitor the implementation of constraints according to the taxonomic granularity design pattern.

The Figure 4.1 shows in summary how the Django system handles data validation. This provides us enough insight helping us to understand how to change the framework, but still play within two philosophies. It shows how the system components beneath the system impedes the research project philosophy implementation. We use an implementation of how Django web framework handles data validation to illustrate the system functioning mechanism clearly with regard to the research project and MVC philosophies.

We evaluated the current capabilities of Django based on the philosophies checklist with regard to Django data validation.

Research Project philosophy

The research project aims at implementing two philosophies.

a) Database consistency

Django design maintains consistency in the database layer by enforcing the inherent constraints defined and generated by the model. It also maintains the explicit constraints in the application layer, though the complex business validation rules defined in Django view module as shown in Figure 4.1.

b) Taxonomic data granularity in enforcing constraints

The Django design system does not follow data granularity order when enforcing constraints according to the constraints enforcement design as stated in Chapter 3. The data first goes through the form data validation that data types. Then the data is submitted to the database for database inherent constraint checks.

MCV architecture philosophy

The MVC design philosophy includes the DRY principle: reusability and code generation by web framework to build applications in a rapid and clean manner.

a) DRY principle

The business rules defined in the view cannot be accessed by other forms in other applications sharing the same database. This implies that the rules have to be copied to similar viewa for each application accessing similar tables in the database. This violates the ‘Don’t Repeat Yourself’ principle.

b) Reusability

There is a lack of reusability of already defined functions since they cannot be called outside the application. This means that other applications have to define new constraints based on their design without breaking the integrity of the database.

c) Clean

The is no clean order followed when enforcing constraints. Some are strewn in the application while others are put in the database as explained earlier.

d) Code generation by web framework

The code is purely pythonic. This resonates well with the purist idea of keeping Django to be pure python. The SQL is wrapped in the ORM to generate schemas and inherent database constraints.

4.2.2 Extending Django to support spatial database consistency

We looked into two implementation solutions for creating spatially consistent web applications based on the proposed design method. These implementation approaches are: model and function validation designs. To achieve this, we gathered an in-depth understanding of how Django and GeoDjango web application framework handle data validation in various modules. We discuss both implementation approaches below.

4.2.3 1. Model validation design implementation approach

In this implementation approach, we extended the Django model to support the definition of check constraints. This involved patching some of the responsible modules like: creation.py, validation.py, options.py and models.py. The model class below shows how the model handles the definition of constraints. The constraints are passed as a set of tuples within a tuple under a variable named constraints as shown in model class Distrib below. This ensures that the check constraints tasks conducted at the view module are moved to the database as shown in Figure 4.2. Therefore, we were able to achieve the task of moving the check constraints that resided in the application layer into the database layer.

However, complex business logic still remained in the form module which is part of the application layer as shown in Figure 4.2. Also, despite having defined the check constraints in the model, we still did not have control of ordering the constraints executions based on taxonomic granularity, such that we begin with attribute constraints, object constraints, then table constraints and finally database constraints.

We were not able to have the business logic entrenched in the database. Therefore, the DRY principle is still not fully adhered to because we only managed to move the check constraints to the model but not the complex business logic defined in the application. This meant that all the complex business logic

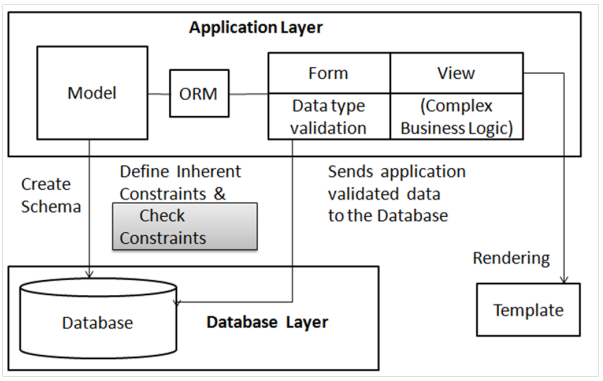

Figure 4.2: Django extension to support check constraints

functions have to be repeated in all the forms that have access to a similar table in the database. Therefore, this approach works on the a compromise of having some constraints reside in the application layer and some in the database layer.

```
class Distrib(models.Model):
    speciesid = models.CharacterField(length=50)
    name = models.CharacterField(length=50)
    geom = models.PolygonField(SRID: 4326)
        class Meta:
            constraints = (
            ('object constraint',Check(speciesid__gte = 100, speciesid__lte = 999)),
            )
```

4.2.4 2. Function validation design pattern implementation approach

In this design implementation, we address the problems that arose when implementing our design method using the model validation approach. We therefore developed a function implementation that would guarantee the validation in the database. To comply with our research project requirement to fully have a system that maintains spatial consistency in the database as well as adhering to the taxonomic granularity order, we developed this implementation for the proposed design method.

The implementation relies on database integrity-preserving functions that wrap and validate any posted data in the form GUI by a client user. Therefore, the business logic will no longer be defined in the view but in the database, different from functions in Figure 4.1 and Figure 4.2 functions. The database integrity preserving functions enable use to define a wanted validation order in the database based on granularity.

Moreover, we had to subject our implementation analysis to see whether it meets the research project's and MVC philosophy. First, the integrity-preserving functions are programmed using server-side PL/pgSQL procedural language

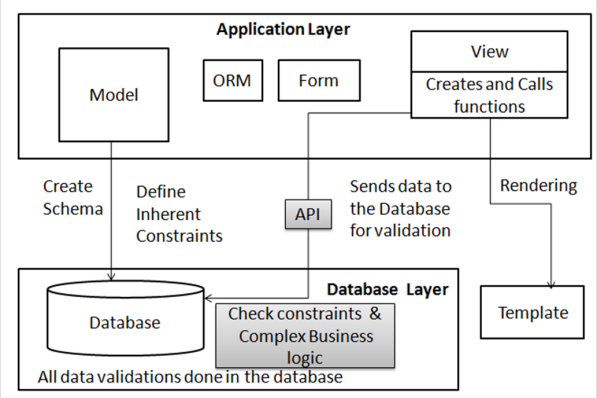

Figure 4.3: Extended Django that supports function creation and Remote Procedure calling

and stored in a database. We also created a function calling API in the model module that calls the functions into the view avoiding the view validation. This makes the implementation DRY principle compliant. The application is clean because the complex coding is taken to the database where the data reside. It is very flexible because the server-side programming is versatile and provides built-in functions.

The created integrity-preserving functions are stored in the database where they can be reused by other applications to access the same tables of the database. This implementation also ensures that uniform integrity rules are enforced by all the applications or other forms that share common tables. It also promotes rapid creation of web applications. This is because once an integrity-preserving function of a certain table has been created, the developers are left with the task of only calling the functions.

It is loosely-coupled because when changes are made in the API, it does not affect the entire system. For instance, each function exist as an independent validation function, that is, any alteration will not affect the other constraints. The function creation is done using raw PL/pgSQL because there is no wrapper in Django for generating PL/pgSQL statements. Therefore, facility of code generation is lacking. However, when a wrapper is developed as discused in Section 3.5 it will make it easy to define the constraints within a Django view module. This will ensure that the implementation grows to become pure Pythonic.

4.2.5 Summary of steps followed when implementing constraint design method

Constraint design method prescribes a four step method to promote spatial database consistency in web applications. The method was developed after taking into consideration constraints design parameters and techniques discussed in Section 3.3.

1. Understand both simple and complex business logic, then define each ba-

sic constraint based constraint granularity levels (i.e. attribute, object, table and database).

2. Then an integrity-preserving function is created that wraps each constraint in a function based on granularity levels. The integrity function return boolean.

3. Then create an update transaction function that calls the integrity-preserving functions. If the called functions return True then update statement commits the transaction into a database.

4. To utilize the transaction update function, we use a Remote Procedure Call API.

It should be noted that if the integrity-preserving functions are implemented using triggers, then the fourth step is avoided. However, because of the limitations of triggers being slow and opaque, we use a Remote function call Remote Procedure Call API.

4.2.6 Implementation of the constraint method to design integrity-preserving function

We show how the proposed constraint and API design methods discussed in Sections 3.3.14 and3.4.2 are used to create integrity-preserving functions and develop Remote Procedure Call API are implemented in Django web application framework and PostgreSQL.

Attribute integrity-preserving function implementation

The design method discussed in Section 3.3.5 has been used to create an integrity-preserving function of attribute constraint granularity level shown below. The integrity-preserving function returns TRUE if a valid genusid is entered but if a wrong genus is entered it raises an exception indicating the nature of the of violation. Algorithm 16 implements the method developed to create integrity-preserving function of type attribute.

Algorithm 13: *attribute_genus*

Require: Create a function $attribute\_genus$ that has parameters $(gdxint)$.
Require: Data to be validated gdx.
Require: Create temporary record called $myrec$.
Ensure: Validation is of type $attribute$.
1: **if** $IF\ myrec.gidx\ IS\ NOT\ NULL\ THEN$ **then**
2: Abort, and send warning to the GUI.
3: **else**
4: **return true**,
5: **end if**
6: function $attribute\_genus(gdx)$ holds when data gdx is valid.

The code below shows how the Algorithm 16 implemented in PL/pgSQL is used to creat integrity fucntion.

```
CREATE OR REPLACE FUNCTION check_attribute_genus(gdx int )
RETURNS BOOLEAN  AS $ $
--The function arguements are represented as  (gdx = genusid)

DECLARE

myrec RECORD;

BEGIN
     -- Searches for the supplied  gidx within the genus table
     -- and then assigns the rows into the myrec record

      SELECT INTO  myrec * FROM genus WHERE gidx = gdx;

      IF myrec.gidx IS  NOT NULL THEN

      -- This statement checks uniqueness of the input data
      -- This is an attribute constraint

         RAISE EXCEPTION
           'Reject as a primary key because % alredy exists', gdx;
      ELSE
            RETURN (TRUE) ;

  END IF;
END;
\$\$ Language 'plpgsql';
```

Transaction update function

The transaction value below is used to update data into genus table in the Amazonia database where the genusid represented as gdx was found to be valid. The update function returns void.

Algorithm 14 illustrates the design method discussed in Section 3.3.9 used to create transaction update function in PL/pgSQL.

```
CREATE OR REPLACE FUNCTION update_genus(gdx int) RETURNS VOID  AS\$\$

-- Performs the transaction and updates  the genusid.

IF (select check_attribute_genus(gdx) = True ) THEN

          UPDATE genus SET gidx = 'gdx' WHERE gidx = '45';
```

Algorithm 14: *Create update function*

Require: Create a update function $update\_genus$ that has parameters$(gdxint)$.
Require: Data to be updated gdx.
Require: Database status G
1: **if** $IFcheck\_attribute\_genus = True$ **then**
2: Update genus table.
3: Database state changes $G \leftarrow G'$
4: **end if**
5: function $update\_genus(gdx)$ holds when data gdx is valid.

```
          END IF



      RETURN;
END;

\$\$ Language 'plpgsql';
```

Remote Procedure Call API implemetation

The Remote Procedure Call API below that reside in Django view.py module was created using the method discussed in Section 11. It is used to post the genusid (represented as gidx) entered at the GUI to the model.py API portion and sends the response received from the database via the model to the GUI.

```
def  genus(request ):


    if  request.method == 'POST':
        form  = GenusForm(request.POST)


      if  form.is_valid():
       my_funct  = Call_Dbase__genfunctions()

       genusupdate = my_funct.call_distrib(request.POST['gidx'],)

          return  HttpResponseRedirect('/distrib/')

    else:
     form  = DistribForm()

    return  render_to_response('form.html',{'form':form,'error':True},)
```

The portion of the Remote Procedure Call API is created using the methods described in Section 10. It calls the validation function 'check_attribute_genus' and transation function 'update_genus'. When the attribute is found to be valid, genus data is updated.

```
class Call_Dbase_genfunctions():
    def  call_genus(self, genusid):
       cursor  = connection.cursor()
       genusupdate   = cursor.callproc('update_genus', (genusid))
       cursor.execute('COMMIT')
       cursor.close()
         return    genusupdate
```

Using Remote Procedure Call API to execute validation and update functions

We use the Remote Procedure Call API to call validation functions and an update function as explained in Algorithm 15.

Algorithm 15: *Using Remote Procedure Call API to execute validation and update*

Require: Create a Remote Procedure Call API.
Require: Data entered at the GUI $gdx \leftarrow gidx$.
Require: Integrity preserving functions $check\_attribute\_genus$.
Require: Call database update function $update\_genus$ to modify database of state G.
1: **if** $check_attribute\_genus(gdx)$ does not hold **then**
2: Abort, Raise warning.
3: **else**
4: Call update function $update\_genus(gdx)$ and commit transaction.
5: Database state changes $G \leftarrow G'$
6: **end if**

4.3 Web application implementation

We are provided with a use case of Amazonia avian distribution data that is used to build a web application. The use case forms the basis to test and evaluate our design method described in Chapter 3. The web application is characterized with an additional Remote Procedure Call API and is built using Django web framework and GeoDjango.

We leveraged the second design implementation as a pragmatic solution to the Amazonia avian distribution use case. This is because we planned to show how database consistency is maintained in the database, and how it promotes spatial database consistency. We also employed it to show how taxonomic data granularity order is used in executing constraint checklist within an integrity-preserving function, ensuring the database integrity is not broken by flawed

data. Finally, the design implementation shows how MVC philosophies of web frameworks are adhered to when implementing the use case.

We elucidate the entire process from installation, to coding and finally having an output in the form of a working web application site.

4.4 Use case motivation

This small document describes a use case for a test on web frameworks. Specifically, that test should demonstrate the ease with which a web-based application can be build that displays some spatial (vector) data functionality. At first, the use case description is simple, and many details are omitted. We expect the use case to gradually evolve and become more complicated.

The use case is a real one, that could be put in place in a project that has been running, slowly, since 2006. The project is a collaboration with INPA, Manaus Brazil on an information platform about Amazonian life forms, i.e., plants and animals and their distribution. At present, only avian life forms are covered, for one because distributions are on average much better known than for other life forms. But over time, the coverage is expected to be extended to aquatic animals like crustaceans, but also mammals, and plants.

4.5 Conceptual schema

In Figure 4.4, is depicted an early and simplified information design of our spatial database. It is derived from an Enterprise Architect project, which is to be distributed together with this document. We continue to describe here what the information content of the system is.

The system captures information about species, here, specifically names in three languages as well as spatial distribution for most (but not all) species. The listname is the scientific name, port_ and eng_name give Portuguese and English vernaculars. For historic and archival reasons, spatial distributions are modelled as separated objects. This decision is to be retained throughout the model.

The system maintains separate objects for avian genera and avian families, with the obvious biotaxonomic structure in place, as depicted in the figure. The system also maintains a single spatial object, which is an administrative Brazilian legal entity, known as Amazonia Legal. This area enjoys special protective measures under Brazilian law, for instance laws about the lands of indigenous people, laws about deforestation, and so on. The purpose of our system is to maintain as faithfully as possible the avian distributions known for the area of Amazonia Legal.

Support information comes from interfluves and ecological regions. An *interfluve* literally is an area between rivers, and for the purpose of mapping species' distributions, Amazonia Legal has been 'cut up' into interfluves. Interfluves are a biogeographic reality in the sense that their borders (the rivers) form natural boundaries for many species. It was later decided to expand the notion of interfluve to the neighbouring areas of Amazonia Legal, both inside and outside of

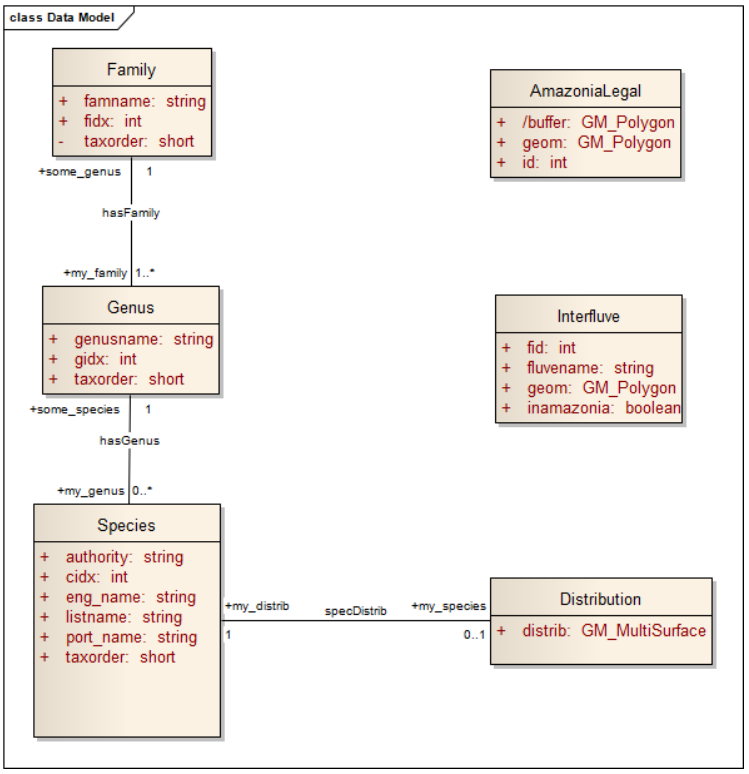

Figure 4.4: Conceptual database schema, version 1

Brazil. The reasons for this will become clear below. Interfluves have a name, a polygon geometry and an indication of whether they are inside Amazonia Legal or not.

Ecological regions have not yet been modelled in the diagram. Also, the default SRS should have been mentioned. Its `srid` is 4326, for all geometries in the information system.

4.5.1 Constraints

The EA project contains a small number of constraints, recorded per class or specifically for an attribute. Some are regular uniqueness constraints (but not, at this conceptual level, primary key constraints, as they surface at the logical level only), others have been informally described in text in the project file.

Observe that there have been defined a few 'unique within' constraints. These specifically apply to `taxorder` attributes. A `taxorder` attribute serves to indicate the linear order within a taxonomy. The taxorder value for a genus, for instance, must be unique for that genus amongst other genera in the genus's family, but not (necessarily) globally across genera.

There is no guarantee that the data that accompanies this case study obeys all the constraints here presented.

4.6 Required functions

In the section below, we describe a small number of important functions that our web application should offer to its users. One philosophy on functional development is to include these as much as possible as stored procedures in the database, so as to make the database a self-contained and functionally complete resource. In the application code outside the database this would be visible as a single function call.

A forthcoming paper will describe and illustrate a method of developing stored procedures for a database, arguing from the perspective of constraint integrity. That method follows an inside-out approach, addressing first attribute constraints, then tuple constraints, table constraints and eventually database constraints. That method is not yet available (or described) but has been discussed with supervisors.

4.6.1 The web application

The primary, web-based, application that we have in mind here is essentially a content management system. The reason is that in such a system data updates are more prominent. We describe a few of these functions. We do describe them as integrity-preserving functions, i.e., they are meant to never invalidate database integrity, and should be implemented as such.

4.6.2 Required database functions

Where details are left out of the description, the implementor can decide at will. An implicit precondition and postcondition is always that the database is consistent.

add_family

Signature	`newfamname:string`, `existingfamname:string`, `beforeorafter:{'a','b'}`
Precondition	`newfamname` does not exist yet `existingfamname` does exist
Semantics	adds a new family to the database just before/after a given family
Postcondition	`newfamname` exists and occurs just before/after `existingfamname`

Remark that, though we describe this function first, it does not actually precisely constitute a consistent database function. The reason is the multiplicity constraint (1..*) on hasFamily, which requires that any family is associated with at least one genus. A new family can therefore only be added if it also has a genus. The challenge is to come up with a stored procedure that combines those two database actions. But initially one can (should!) relax the condition and assume (0..*) multiplicity.

add_genus

Signature	`newgenname:string`, `existinggenname:string`, `beforeorafter:{'a','b'}`
Precondition	`newgenname` does not exist yet `existingfamname` does exist
Semantics	adds a new genus to the database just before/after a given genus, and in that given genus's family
Postcondition	`newgenname` exists and occurs just before/after `existinggenname`

This is a robust database function.

add_species

Not fully described for the moment. Provides values for all attributes but `cidx`, which will be autogenerated, and `taxorder`, which will be derived from providing (as above) an existing species by `engname` or `listname`, and a before/after indication. This can be done in two functions, one for `engname` one for `listname`, or in just one of these. Obviously, the various uniqueness constraints should remain valid.

add_distrib

Adds a geometry for an existing species. The specific challenge here is to obtain the geometry from an indicated shapefile. An extra version of this function will allow on-screen digitization of the geometry. In both cases, the stored geometry will be restricted to or by `AmazoniaLegal.buffer.bbox()`.

change_restrict_distrib

This function changes the distrib for a given species by restricting the old distrib to a given list of interfluves. The latter are indicated by `fid`, but the web app may provide a picklist, possibly even through a map interface.

change_augment_distrib

This function also changes the distrib for a given species, but now by filling up boundary slivers of an indicated interfluve and the old distrib. An extra parameter indicates the size threshold of slivers repaired, above which slivers are left untouched.

More functions will eventually be described here.

4.7 Constructing the Web GIS application

We elucidate the entire process from installation, to coding and finally having a final output in form of a working web application site.

4.7.1 Django web application framework setup

All the software package packages constituting the web framework setup are open source, this gives us the rights to access the source code and improve it to suit the web application requirements. We started first with acquiring the software, installing and configuring.

Python, Django and GeoDjango installation

To begin, version 2.6 of Python was installed in the main server to support pythonic Django web framework and its spatial extension GeoDjango. The Django web framework cannot run without the presence of the Python language in the computer environment that the application is meant to run on. Then, version 1.1 of stable Django was downloaded and installed systematically following the installation procedure provided at the official Django official site. We also downloaded and installed GeoDjango as add-on spatial package for Django. Django and GeoDjango are both installed in the server that hosts the web GIS application.

Spatial libraries

GeoDjango comes with Proj4, GEOS and GDAL/OGR spatial modules. Proj4 is a cartographic projection library. Geometry Engine Open Source (GEOS) is a library that facilitates programing of 2 D geometries. The GDAL (Geospatial Data Abstraction Library) library has tools for manipulating raster data while OGR facilitates manipulation of simple feature geospatial vector data.

Web server setup

Django comes with its built-in server for development purpose, meant for small capacity requests. However, for online deployment to a lager number of distributed users, it uses Apache web server which is capable of processing and responding to a large number of client requests for a busy site. Hence, the Apache web server and mod_python were installed and configured to serve the web clients. Mod_python is a Python interpreter embedded in Apache that facilitates writing of fast Python web applications that run faster than traditional CGI.

Database connectivity

The web application being dynamic and interactive, it needs a repository to enter, modify and store data from active users. Therefore, we also had to download and install the PostgreSQL, a database that supports spatial geometries. Moreover, in order to have more predefined functions in the database we also installed PostGIS which is a spatial extension of PostgreSQL. To facilitate connectivity between the web application and the database, psycopg2 was installed.

4.7.2 Creating a spatial database

With installed software up and running, a spatial database which is core of our research study, is the next initial part of web application creation. The spatial database was created by running the command prompt below in the home directory of the user with rights to create databases. This is due to the fact that the setting.py file as shown in Appendix A in Django application will request for the database during configuration of database connectivity.

- `C:/Users/postgres/ createdb -T template_postgis Amazonia.`

All the tables/schemas created either through ORM or directly in the database are hosted within the Amazonia spatial database.

4.7.3 Building Amazonian web GIS application

With the spatial Amazonia database ready, we created a directory called Brazil where all the application files will reside. This file can be located anywhere in the computer but not in the root directory for security reasons. During the project creation is placed automatically under the PYTHONPATH.

Project Creation

In the directory called Amazonia. we ran the command below to create a project called Brazil that contains modules to manage and configure the web application.

- home/Amazonia/ django-admin.py startproject Brazil

The Brazil project is a Python package directory, the command above adds it to the PYTHONPATH to enable its module files to be called by any Python module file. In addition, the above command prompts the web application framework to create automatically the project modules the include: _init_.py, manage.py, settings.py, settings.py and urls.py.

The roles of the Python modules which are python files are explained as follows:

init.py

This module informs the Python interpreter that the Brazil directory is a python package. That is, an extension of the web application framework.

manage.py

It is a thin wrapper around django-admin.py, it executes two roles before forwarding to django-admin.py [12]:

1. It places the created projects package on sys.path.

2. It configures and directs the DJANGO_SETTINGS_MODULE environment variable to point towards the created projects settings.py file.

The django-admin.py is a command prompt utility for administrative tasks in Django.

settings.py

The all web application settings and configurations are performed in this module. Some examples of configuration include

urls.py

Application urls are defined in this module. The url.py render the web application to the browser depending of how it has been described by a rendering functions in the view.

4.7.4 Creating the web application

We then ran the command below within the project Brazil directory to create the web application fundamental modules.

- home/Amazonia/Brazil/python manage.py startapp Avian

The modules created included model.py and view.py,.

model.py

This module is used to define the table schema, multiplicities and implicit constraints. It is in this module that part of the API code for calling database defined functions is situated as stipulated by the discussion of the system design method. Admin and GeoManager(a class that manages spatial geometries) functionality are also called from the model module. Model.py module is shown in Appendix B.

view.py

This module constrains the controller functionality as stated in the literature review of Section 2.2.3. Also some of the database API code for calling integrity-preserving functions is defined is this module. The API takes the data validation role from the view. The view receives the data from the form field that is entered by the user and passes it to the called function for validation. Once the submitted field data is considered to be valid, the API commits it to the database. The Appendix D shows the view.py module.

A number of applications can be created within the Amazonia project in an a similar manner by observing the web application creation procedure.

4.7.5 Configuring the application

Within the setting.py module, shown in Appendix A, the following configurations were performed:

Configuring the database

The database configuration was done by entering the database engine name, database name, database user, host, password and the port. The database engine used in our scenario is the postgresql_psycopg2.

Configuring the base directory

Base directory is a variable that defines explicitly the path of the project directory. This path locates the template that determines how the site is rendered. The path also contain other addition visualization widgets and raw vector data like shapefiles. Appendix A shows an example of how we defined our base directory.

Installed applications configuration

The modules that are needed to have the application up and running are defined in a python tuple INSTALLED_APPS. For instance 'brazil.avian' calls the Avian application, 'django.contrib.gis' calls the geospatial functionality, 'olwidget' calls JavaScript OpenLayers, 'django.contrib.admin' calls the built-in admin functionality, the rest are built-in Django ORM functionals that participate in creation of schema and authentication of users of a site.

```
INSTALLED_APPS = (
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.sites',
'django.contrib.admin',
'django.contrib.gis',
'brazil.avian',
'olwidget', )
```

The entire setting file is listed in Appendix A.

4.7.6 Creating the database tables

The Model.py module supports the creation of schemas and definition of inherent database constraints. It also supports the creation of table schema in the database and simply informs the web application of their existence by using legacy functionality to automatically write back into the model. According to the use case provided, UML modeling language was used to define the conceptual schema, which we later translated into the logical design and finally after some modification into a the physical database.

In our approach, we experimented by defining the table in the database. Figure 4.4 shows the conceptual design of the database based on the user requirements. Logical design phase was run and the physical design which was in the form of SQL statements. The correctness of the SQL statements were looked into critically and additional commands were added. For instance, UML does not support the creation of spatial geometries column in a schema, this was done manually.

After ensuring that the database tables have been created and are well defined interm of fields, multiplicity and inherent database constraints, the entire tables are mapped backwards into Django as using Django Legacy tools. The command below was run in the Brazil directory.

- home/Amazonia/brazil/ manage.py inspectdb
- home/Amazonia/brazil/ python manage.py inspectdb >home/Amazonia/brazil/ avian/models.py

Thereafter, we checked and corrected any errors committed in the process of database introspecting and mapping into the model module. Appendix B shows the model contents inherited from the database and translated into the model.

Synchronizing the application with the database

Once the model was found to be clean of errors, we ran the 'run synchronization' command below that created other Django schemas in the database. We avoided the duplication of the mapped model schemas by commenting out the 'brazil.avian' in the setting.py module.

- Amazonia/brazil/ python manage.py syncdb

Thereafter, we uncommented the 'brazil.avian' and started the server. At this point the database schemas were considered complete and ready for data entry, unless when minor glitches were reported, then then edits were conducted.

Remote Procedure Call API

This API is designed to facilitate calling of functions by the Django view (controller). The view receives the data data it receives from the form entry fields and forwards it to the integrity-preserving function as parametric arguments. Therefore, the function wraps the data and then subjects it to the database layer for consistency validation. If there exists any error, the API forwards the error to the Template (viewer) for the user to know the kind of violation committed. The API also allows for insertion of the name of the function to be called. The function is inserted at the API code located in the model as shown in Appendix B. The API resides partly in the model module as shown in Appendix B for easy entry of names of functions without having to interfere with the main web framework code. The API inherits classes and methods within the web framework like the transaction and connection classes. This optimizes , and help in making the API code making it lean, efficient and effective.

Creating the form fields

The form module creates the GUI fields that are to appear in the template. The form either reads and inherits from the model entries columns, or can be defined explicitly from scratch. In our scenario, we defined explicitly the form contents because we added an OpenLayers widget for drawing the geometries interactively. The OpenLayers widget fields requires for explicit definition of parameters as shown in Appendix C to set up how OpenLayers is rendered at GUI. The other reason for using this approach is that not all tables created need to be modified, some just serve as base data referenced by those being modified, hence no need to have their fields in the GUI forms. Appendix F has the code of the form module used to render the OpenLayers widget and the form fields in the GUI.

Creating the Django view

The view in Django plays the role of controller in the MVC architecture as stated before in Chapter 2. It facilitates communication between the model and form modules and the template (viewer). It also accommodates method definition for data integrity validation, that is, when data is entered via the the GUI it must first validate the data before posting to the database. The business logic is defined and enforced within the application methods, hence the term 'application layer consistency enforcement'. The method normally lacks capability to define complex spatial constraints.

However, in the new design approach, we no longer have of the application layer validation by calling the Remote Procedure Call API situated in the model module. The API ensures that the data is forwarded to the database for validation, where all our constraints reside. If the data is found to be correct, it is committed to the database and saved, or else it raises an exception prompting the user to enter correct data.

The view is also used to render the database data into the template for visualization. This ensures separation of concerns that sets aside the core programming code from the aesthetic styling CSS, HTML and JavaScript for presentation and visualization. The view code is displayed in Appendix D.

Creating the template

The template is the styling that determines the look of the web application in the web browser. The template can be in HTML, CSS, JavaScript or XML. The widget as used in our use case is embedded in the HTML template as shown in Appendix F. The templates files reside in the Amazonia project directory and its path is defined as shown in Appendix A in the setting file under TEMPLATE_DIRS.

Configuring the OpenLayers widget

This widget enables drawing of geometries interactively like in a GIS environment. Base maps can be embedded beneath. We use the use case data as our

base data. We created a folder called OLwidget in the Amazonia project directory. The location of the widget is configured in the setting module under MEDIA_ROOT, MEDIA_URL and ADMIN_MEDIA_PREFIX as shown in Appendix A.

Configuring the urls model

The urls determine the paths followed to allow a request reach the web application as shown in Appendix E. The responses are also relayed through the same path. The url depends on the rendering function defined in the view module.

4.7.7 Loading the use case data into the created tables

GeoDjango has a class called LayerMapping that is used to load OGR vector files like shapefiles, into the created models that support geographic data. The other easy way is to first import the use case data into Amazonia database using built-in PostGIS functionality, which creates tables automatically. Then using appropriate SQL INSERT AND UPDATE statements the created schemas, are populated thereafter. In this implementation, we used the first option of LayerMapping to upload the use case base data into the schemas.

Setting up the admin

The admin is a built-in capability of Django that enables the administration of the site. It is used to create users and the security level of each user. It also generates GUI form fields interface the emulate the model field structure used for data entry. GeoDjango has its built-in OpenLayers that is automatically generated in the edit map GUI.

However, in our design we did not use the admin auto generated GUI because it is complex to tweak due to lack of documentation on how it operates. Therefore, for us to have full control to define the GUI to suit our user needs, we programmed the GUI from scratch rather than obtaining it from autogeneration. This act enabled us to create a link between the GUI for data entry and the called functions and the database validation report. The validation report can either return success after submission, or a failure indicating where the error was committed.

Up to this stage, the application layer is considered complete. The next phase is the definition of business rules using the PL/pgSQL database programming language. The wrapper does not exist, so we pass raw PL/pgSQL into Django.

4.8 Implementation of the Amazonian use case

The constraint design method and the prototype architectural design discussed in Chapter 2 were applied to construct the Amazonian GIS web application.

4.8.1 add_distrib function

Adds a geometry for an existing species. The specific challenge here is to obtain the geometry from an indicated shapefile. An extra version of this function will allow on-screen digitization of the geometry. In both cases, the stored geometry will be restricted to or by `AmazoniaLegal.buffer.bbox()`.

Algorithm 16: *Create add_distrib function*

Require: Create a function $add\_distrib$ that has parameters (distribgeom geom).
Require: Data to be validated $distribgeom$.
Require: Create temporary record called $myrec$.
Require: **Let** V = $SELECT\ geom\ FROM\ amazonialegal$
Ensure: Validation is of type $database$.
1: **if** $(Select\ ST\_Intersects((distribgeom), (\mathrm{V})))IS\ NOT\ NULL$ **then**
2: Abort, The speciedid $distribgeom$ is not in the Amazonlegal.
3: **else**
4: **return true**,
5: **end if**
6: function $add\_distrib(cidx)$ holds when data $distribgeom$ is valid.

Algorithm 17 calls the $add\_distrib$ integrity-preserving function and if the called function returns True, then the database is updated with the new distrib geometry.

Algorithm 17: *add_distrib update function*

Require: Create a update function $update\_genus$ that has parameters distribgeom geom).
Require: Data to be updated $distribgeomgeom$.
Require: Database status G
Require: Call function $add\_distrib$
1: **if** $(SELECT\ add\_distrib(distribgeom)) = True$ **then**
2: $INSERT\ INTO\ distrib\ (distribgeom)\ VALUES\ (distribgeom)$
3: Database state changes $A \leftarrow A'$
4: **end if**
5: function $update\_genus(distribgeom)$ holds when data $distribgeom$ is valid.

4.8.2 change_restrict_distrib function

The Algorithm 18 is used to creates an integrity-preserving function $restrict\_distrib$ that restricts a species to certain interfluves before making an update.

4.8.3 change_augment_and_restrict_distrib function

The update function in Algorithn 19 calls and performs $restrict\_distrib$ function which checks that a species is with a certain interfluve before making an

Algorithm 18: *Create change_restrict_distrib function*

Require: Create a function $add\_distrib$ that has parameters (distrib-geom geom).
Require: Data to be validated $J \leftarrow distribgeom$.
Require: Create temporary record called $myrec$.
Ensure: Validation is of type $database$.
Require: **Let** S = $SELECT\ geom\ FROM\ amazonialegal$
1: **if** $(Select\ ST\_Intersects((\mathrm{J}), (\mathrm{S})))\ IS\ NOT\ NULL$ **then**
2: Abort, The speciedid J is not in the Amazonlegal.
3: **else**
4: **return true**,
5: **end if**
6: function $add\_distrib(cidx)$ holds when data $distribgeom$ is valid.

update. The update ($update\_augment$ and $Restrict\_distrib$) function has smart code that repairs the slivers before doing an insertion once ($update\_augment$ and $Restrict\_distrib$) returns a boolean of type True.

Algorithm 19: *Create update function* ($update\_augment$ *and* $Restrict\_distrib$)

Require: Create a function $update\_augment\_distrib$ that has parameters (distribgeom geom fid int).
Require: Data to be validated $distribgeom$.
Require: Create temporary record that store fid.
Require: Database status C.
Require: **Let** U = $SELECT\ GeomUnion(geom)\ FROM\ interfluves\ where\ id = \ fid$.
Ensure: Call all the integrity functions $change\_restrict\_distrib$ and run them within from update function.
1: **if** $((SELECT\ change\_restrict\_distrib)\ and\ (change\_augment\_distrib) = True)$ **then**
2: **Update** $(Select\ ST\_Intersects((distribgeom), (U)))$
3: {Repair any slivers with a zero threshold from the interfluve boundary, i.e. uses the boundary to fill}.
4: Database state changes $C \leftarrow C'$.
5: **end if**

4.8.4 Amazonian web application layout

The web application GUI is divided into two section. One for entering and editing data and the other for visualization of the existing datasets by the users.

Figure 4.5 show the Amazonia avian distribution home page. A user can modify a tables in Amazonia database by clicking at an appropriate hyperlink. For instance, to add a distribution geometry of species into the distrib table a user needs to click 'Add avian distribution geometry' as shown in Figure 4.5. It provides for entry of a speciesid and drawing of geometrical features. If a user enters a geometry that falls outside the Amazonialegal a warning message is

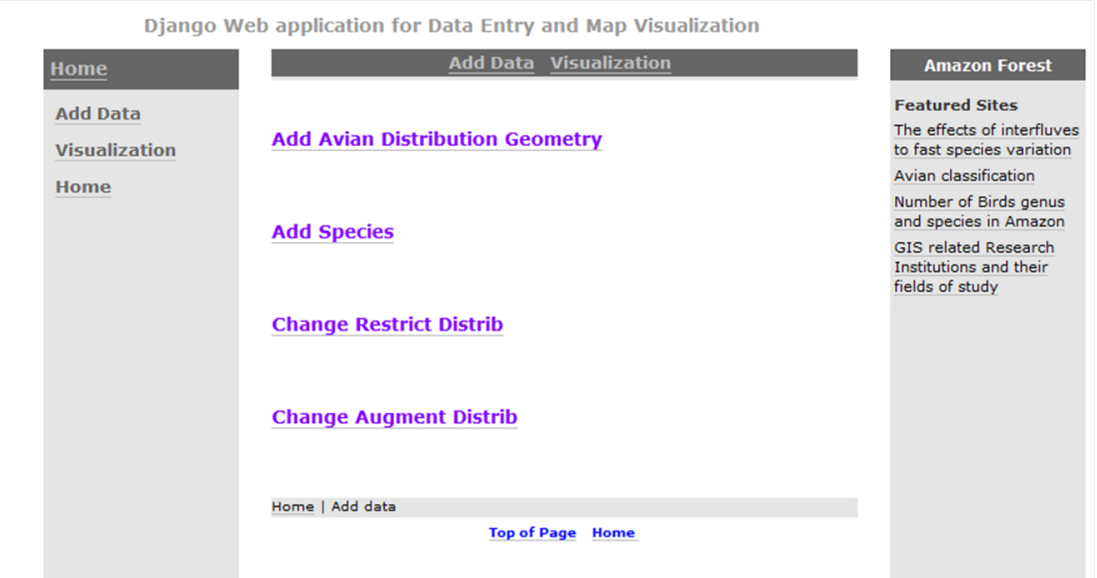

Figure 4.5: Amazonia web GIS home page

sent to the end user indicating the nature of the violation as shown in Figure 4.8 but if the geometry and speciesid entered are correct then the distrib table is updated.

Figure 4.6 shows the form field for entry data into the species table. It does not according to the use case it only takes attribute data.

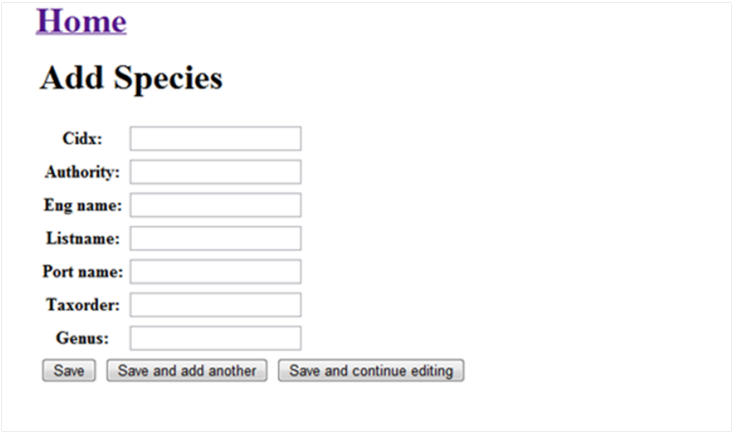

Figure 4.6: Add species

Figure 4.7 shows the form field of the distrib table which according to the use case has an attribute field for speciesid and an editable geometric field. At the bottom corner of the editable map is text box that shows the coordinates of the entered featured.

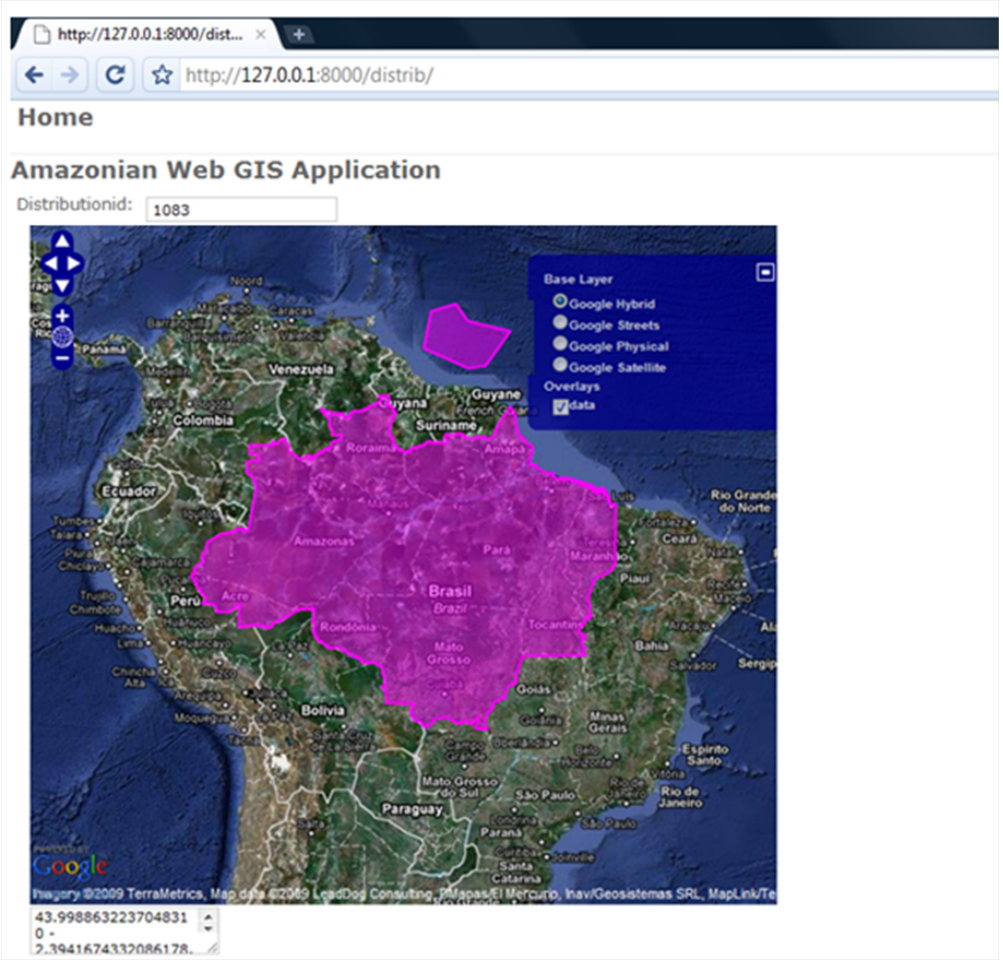

Figure 4.7: Amazinia Legal

Figure 4.8 shows the error message sent to the user if s/he enters erroneous data. For insatance, a user entered wrong geometry outsidet the amazonia legal for a species of id 1083.

Figure 4.8: Warning Message

Figure 4.9 shows an info box at the visualization section that gives information about a feature that a user has clicked on.

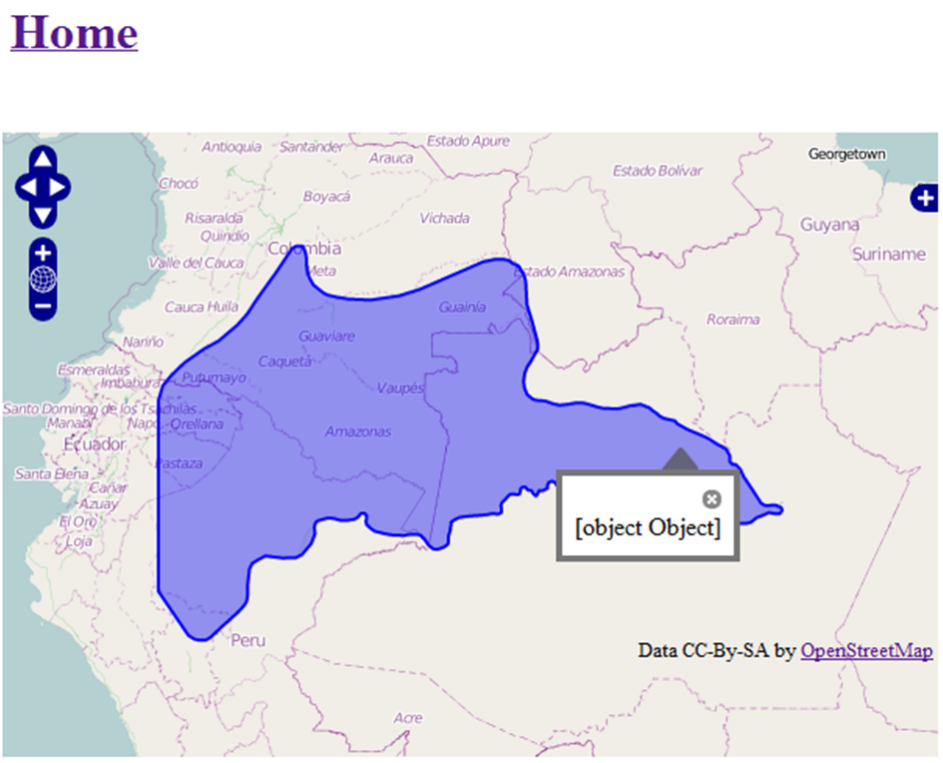

Figure 4.9: Get info

Chapter 5

Discussion, Conclusion and Recommendation

5.1 Introduction

Software systems have life cycles that they undergo in development. This cycle may often take a long time and a considerable amount of human resource before considered complete, but it never reaches absolute completeness. This principle of continuous improvement also applies to web application frameworks like Django. Therefore, development of web frameworks does not stop at the release stage. Often, users of web frameworks have to tweak them to suit specialized needs, address technical problems or correct code glitches. In our research project, we extended Django to support the maintenance of spatially consistent databases by leveraging the strengths of hosting the business rules and other constraints within the database layer.

In this chapter we discuss what has been achieved so far in terms of knowledge of web frameworks and implementation of the constraints design method and prototype architecture to meet the set out objectives. We also recap on how the formulated questions were addressed. Moreover, we assume an unbiased perspective of criticism to our research work. This will finally pave way to raise new questions and to build cogent arguments to back our proposed recommendations for future work.

Django as a collection of libraries used to rapidly build clean dynamic web applications, keeps changing day by day to suit a wide range of users. Django web applications handle a collection of their contents and services that can either be for personal, public, business, scientific, or user group data and can be represented as raster maps, vectors, graphical image, audio,video, digital text, or geoprocessing functions. Django, as displayed in the implementation stage, can be used to build a digital library of spatial data from various distributed users for diverse purposes, in which collaboration and data sharing are important social elements.

5.2 Discussion

We discuss the research project findings with regard to maintenance spatial database consistency in web application frameworks.

5.2.1 Research study overview

We began the research by first venturing into understanding fundamental architecture of the web application frameworks. We reviewed a number of existing frameworks to understand their architecture and how they handle spatial database consistency. We found out that most web frameworks were mainly designed for non-spatial task like running business enterprises, on-line point of sales, social networks and keeping inventory. The issue of geographical support mechanism was missing in most web framework studied. We looked into Ruby on Rails, Web2py, Pylons, Struts, Drupal and Django. Django was the only web framework in our study that had a fully developed extension that supported the building of spatially enabled web applications.

After delving deep into the architectural design of web application frameworks, we developed an in-depth understanding of how created web applications maintain database consistency. Most web applications maintain implicit and explicit constraints that are used for data validation within the application layer and some partly in the database. For instance, Ruby on Rails, views the database as a storage repository for storing data while all the constraints reside in the application layer. Django and its spatial extension GeoDjango, that formed the case study of our research project, was found to maintain implicit and explicit spatial constraints in the application layer and partly in the database. The inherent constraints like primary keys, referential keys, the geometry type and the SRID are stored in the database. However, the user-defined business logic resides in the application layer, defined as Django functions within the view module.

The built-in and user-defined form classes of Django are in charge of data validation. Forms are part of the clients GUI. Form receives posted data by a user and forwards it for validation within the functions in the application layer. The investigation found out that the Django design contravened the DRY principle of MVC architecture design. There is a non-conformance of the DRY principle because the views that access the same table in the database had to copy the same validation code in each view. This compromises the spatial integrity of the database because multiple applications can have different coding and small bugs in the code can introduce flawed data into the database.

After understanding how the web application frameworks operate with regard to maintaining spatial database consistency, we had to come up with new constraint design method based on the research study philosophy as well as the MVC design philosophy. The research study advocates for two main philosophies which are: to maintain spatial consistency constraints within a database and to enforce constraints using a checklist based on constraints taxonomic granularity levels. The MVC architecture philosophy includes: 'Don't Repeat Yourself' principle, loosely coupled, reusability and code generation for rapid,

clean creation of web applications. The design method and its implementation had to consider all these philosophies.

During the implementation of the design method to extend the Django framework, two school of thoughts were conceptualized and developed. First, was the model validation approach that aimed in migrating the constraints defined at the form (e.g. data type validation) and view (i.e. complex business rules) module at the application layer to the model layer that would then be translated into the database via the ORM, as explained in Section 4.2.3. In this approach, we were able to only achieve check constraints migration hence leaving a large portion of the complex (i.e. conditions that involved spatial evaluation expressions) business logic still defined in the application layer.

To correct the limitation in model validation implementation, we came up with a second design implementation called the function implementation approach that took away all constraints from the application layer to the database layer, as shown in Section 4.2.4. These functions would be used to wrap input data, and validate them based on constraint taxonomic granularity before doing an update, deletion or an insertion of data in a database. The function validation approach complies with the research philosophy of having all the constraints reside in a database and following the taxonomic constraint granularity. It also complies with the MVC architectural philosophy of the DRY principle, loosely coupled and rapid and clean application development. This is because the functions are located centrally in the database and all applications that modify the database are exposed to the same standard of integrity control.

Within the integrity-preserving function, complex database constraints can integrate with smart code that does preprocessing before the data is stored in the database. For instance, a smart code can be used to limit a drawn species geometry to certain interfluves that support them. If the geometry overshoots, the function carries out the slicing process before saving the correct geometry. This introduces smart checks that never existed in spatial databases or desktop GIS applications.

To test and evaluate our design implementation, we used the Amazonia avian distribution use case and dataset provided to create an Amazonia GIS web application. The Amazon web application is divided into two main sections one for data modification and another section for visualization. We were able to call functions (using Remote Procedure Call API) that validate constraints from the database into Django web framework and passed the data into the functions for validation before a change in database status. After studying and identifying the factors that affect spatial consistency as discussed in Chapter 3, we were able to show that spatial database consistency is achieved by defining the constraints in the database where the data reside. The functions can be called by multiple applications, modifying similar tables in a given database. Enrichment introduced into the prototype shows how web application frameworks like Ruby that lack database layer validation can leverage the function validation approach to ensure spatial database consistency.

5.2.2 Results of the research

During the research project we came up with four findings that constitute our research project results.

1. Constraint design method

The research project developed a four-step constraint design method that promotes spatial database consistency in web application frameworks, as summarized in Section 4.2.5. The design method starts with the identification of constraints into granularity levels from the business rules, then creating integrity preserving functions, then creating update functions, and finally implementing the constraints using the Remote Procedure Call API. The method ensures all constraints reside in the database.

We were able to show how a technique of designing constraints, based on granularity levels, works well with respect to spatial database consistency. The constraint design method enables one to implement constraints in an orderly manner. Granularity constraints, defined as integrity-preserving functions that return a boolean value can be called and reused by transaction update functions and be used to validate data in web application promoting the DRY principle, loose coupling and reusability.

2. Web framework spatial database consistency cube

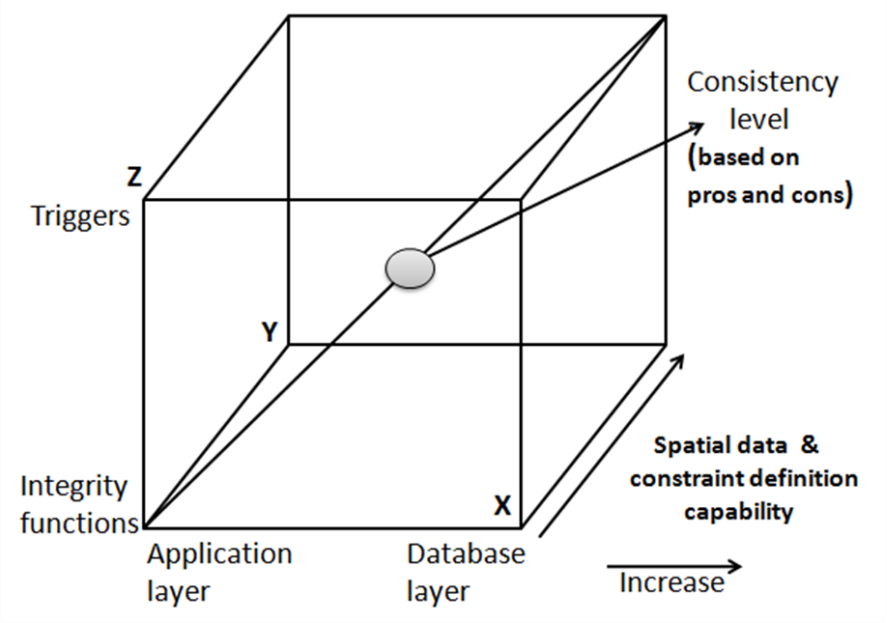

Figure 5.1: Spatial consistency cube

There is a number of factors that affect spatial database consistency in web application frameworks. Figure 5.1 shows a spatial database consistency cube

with three axes, each representing factors that affect spatial database consistency in web application frameworks. The x-axis represents where constraints are maintained, the y-axis represents web application framework capabilities, and the z-axis represents implementation techniques of constraints. We assume that the cube applies if different types of web frameworks under fixed choice of database platform, for instance, PostgreSQL with POSTGIS. Each of the factors has its advantages and disadvantages, as discussed earlier.

When a developer intends to improve the performance of a web application, or develop a spatially consistent web application, s/he should consider these factors. For instance, if one intends to increase the validation speed, then triggers should be minimized. If one intends to expose all data that is entered into the database from different applications to similar validation standards, then database layer validation should be used as discussed in Section 3.3.3. Moreover, one cannot have spatial consistency if the framework does not support the creation of spatial geometries. Therefore, the web framework needs to be extended to support geographical data types.

3. Altering the system component introduces factors that promote spatial database consistency

Frameworks that support spatial data but do not have well-developed built-in libraries to define complex spatial validation constraints, can call and utilize the strengths of databases in creating complex user-defined validation rules. Therefore, to achieve this, a web application framework can be extended to have a Remote Procedure Call API, and also wrap embedded SQL code to enable definition of integrity-preserving functions from the web framework environment.

4. Remote Procedure Call API does not compromise the MVC architecture principles

The use of an Remote Procedure Call API does not affect the MVC architecture principles, that is, the strengths of MVC like DRY principle, code generation and loose coupling still hold. The API enriches the web framework as discussed in Section 3.6.

5.2.3 Achievement and real world applications

There are instances when users are deliberately biased when feeding data into the system for selfish reasons. Mostly, a biased user would want to sway a pattern or trend extracted from the data during data mining and discovery to suit his/her needs. Nevertheless, well-designed consistency constraints and preprocessing with a smart algorithm allows to separate valid data from flawed data, and processing by comparing the consistency of supplied data with existing data according to business rules. The constraint is capable of reject outliers, or flawed data or do internal adjustments. Therefore, all the stored information in the database is subjected to checks that makes it have embedded meaningful traits, beside the geometric and attribute properties.

The quality aspects of data are pivotal in determining the usability of data. The integrity-preserving functions can also be defined to enforce quality parameters like positional and topological accuracy if stated in the business rules. Adding quality checks to stored spatial data provides added value.

5.2.4 What is missing so far, and is not yet achieved

We ventured into the initial process of creating the PL/pgSQL wrapper that maps PL/pgSQL code into Django Python functions that would enable pure Django to generate PL/pgSQL statements. This is to enable the creation of integrity-preserving functions as methods of a class granular_constraints. As the class grows rich in code, similar code can be inherited and extend to create other integrity-preserving functions. The Django created functions in the view module would create database functions through the PL/pgSQL wrapper. This will still comply with the research philosophy that ensures that the business logic still reside in the database layer.

However, because of the complexity and vastness of the Django and GeoDjango code coupled with lack of clear documentation describing the code, we were not able to have a functioning wrapper for PL/pgSQL. With a PL/pgSQL wrapper in place it would remove the need to run raw PL/pgSQL code in Django. It will also promote the use of predefined PostGIS functions to be used to define smart functions for evaluating and validating input data before update, insertion or deletion into or from the database. This is a major research frontier in the Django framework. This is because this functionality is lacking in Django. It will create a new opening for GeoDjango as well, because GeoDjango users will be able to manipulate existing spatial built-in codes to create complex application that guarantees spatial consistency of data.

5.2.5 Critical analysis of the research work

Django uses non-standard terminology when referring to the architectural design. For instance, Django uses the Model-Template-View (MTV) to mean Model-View-Controller (MCV). The use of different terminology may be confusing to the readers or developers who are used to the well-known MCV design pattern.

There is still a limit to the size of spatial data that can be rendered on a web page without straining the system. Large dataset that display well on desktop applications fail to display in web applications that use OpenLayers due to bandwidth and the processing speed of the computer. This is even made harder for mobile devices like PDA and phones that have low processing capabilities. This limitation greatly hinders the interactive mapping and visualization at present. This is the reason we had to split our web 2.0 application into data modification and presentation GUI. We also had to filter what was needed and can be accommodated by the web application at a single moment. Finally, we never tested the created web application on whether it could run effectively on mobile devices like phones and PDAs.

The solution focuses mainly on PostgreSQL spatial database neglecting other spatial databases like Oracle. This is so because we are working on a prototype.

However, with Oracle database the integrity-preserving functions are created using PL/SQL server-side procedural language. The design architecture explained in Chapter 3 still applies. Nevertheless, enterprise databases like Oracle are not available to for free. It requires investments to purchase and support provision, hence, it is not a software of choice for small web applications. On the contrary, PostgreSQL and PostGIS is the chosen platform because it is available for free.

Inasmuch as the integrity-preserving functions can be called by other web applications created by other web frameworks, the design method is specific to Django system components. This implies that applications built by other framework like Drupal or Zope will require tailor-made design to address the problem of spatial inconsistency. This is because each framework handles data validation differently. Others are easy to customize but may have complex algorithms. Lack of standards in web frameworks with regard to maintenance of spatial database consistency makes it difficult for developers to share and improve web framework designs and performance.

For a developer to achieve the same level of result, s/he has to dive into each web framework design to have an insight on how to customize a given web framework. This would have been made easier with the existence of standards. Standardization of the web application framework would have made it easier to develop them based on agreed design guidelines. This would have created a pool of common users working and sharing knowledge in common fields, like database consistency, expanding the user groups of developers. Hence, it would have enabled the web application frameworks in various platforms to interact seamlessly and share easily databases that have constraints at the database layer.

Our implementation design works with a single database that is accessed by the created web application. However, it would be good if the implementation would have factored into multiple databases and applied the research philosophy with regard to having a spatially consistent databases. The design method only calls integrity-preserving functions that reside in a database that the web application is connected with, according to the settings defined in the setting module as shown in Appendix A. With an ability to access multiple databases it would make it easy to access other databases with an intention of accessing more functions and extend the scope of data validation through referenced tables.

Inasmuch we wanted to have all constraints in the database, it is impossible to get rid of double validation that is created as a result of GUI form validation of data type and user-friendly interface properties like drop-down list that select some field data from reference tables in a database. For instance, it is easier for a person working in the field to select than to type a name. The form also does the data type checks to ensure that a submitted value's data type is text and not numeric.

Triggers cannot be gotten rid of completely because inherent constraints like check constraints and referential keys are based on trigger mechanism. Triggers are also reliable in doing back-end data modification after an update to promote database consistency.

The Django web framework is not portable like other web frameworks such as Joomla, web2py and Drupal. This is because it has to be installed in the Python program file. The dependency on the Python program makes it hard to move the application to other computers or development environment in a portable disk with need for reconfiguration. Therefore, the developer will always have to work in the same hardware, where the Django framework is installed, this limits the flexibility in development environment.

When creating a Django project, the web developer interacts directly with the generated Python module codes that exposes it to easy tampering with then. If there would have been a user interface that project the raw code and provides for the developer to have a editor. This would make the work of the programmer easier and faster. The use of command prompt is not user friendly when running generation of codes and starting the database.

The selection functionality in OpenLayers did not enable us to select the database attributes and also see them interactively in the visualization part. We were able to visualize the rendered data as geometrical shapes, but we could not see the attributes of the table content generated during the query process.

We cannot be able to determine to what degree exactly one factor is better in promoting spatial database consistency than the other. For instance, we cannot give a definite value in percentage of how integrity-preserving functions are more efficient compared to the use of triggers.

The web GIS interface created has limited cartographic properties and functions compared to that of a desktop web application. For instance, it is hard to create a map that meets cartographic requirements from the data that is already spatially consistent in the database. This is due to a lack of functions that enable map creation with title, legend, graticules, scale and margins. Also, it is hard to print a map from the GUI. This only makes the OpenLayers suitable for visualization of the datasets. This limitation inhibits the realization of a full-fledged GIS web application, despite a spatially consistent database.

The web application created using Django has a limitation of rendering 3D datasets. Due to the reliance on the Geometry Engine Open Source (GEOS) library that facilitates programing of only 2D geometries. Also, the OpenLayers lack the shading and tinting effect that bring about 3D visualization. The 3D visualization can only be achieved through draping of the vector data over Google map terrain base data. However, creation of Triangulated Irregular Networks (TIN) is not supported by the web GeoDjango functions. Therefore, online display of a 3D spatially consistent database is not currently possible.

When working with the OpenLayers interface, we had a problem with the delete function for geometrical fields. It is not possible to delete specific features from multiple features drawn. If one presses the delete button after making a selection of an unwanted feature, it ends up deleting all the features in the map edit including the ones that are not selected. This makes editing of features interactively hard because this implies that all the geometric features for that transaction have to be created afresh again. The edit mode also lack the snap feature that enables the creation of contiguous features. Most of these operations can be resolved by using built-in functions or defining functions that would handle this edits automatically, based on given intelligent business rules.

5.3 Conclusion

The increase in Internet use has led to the development of interactive web applications that are christened web 2.0. Web 2.0 applications are dynamic, and as a result require a repository to store and retrieve data supplied by active users distributed across the globe. The databases play the role of storing data that can be accessed later through report generation and running queries. The database administrator will always ensure that the data managed in a database is correct and reliable for making informed decisions. For a flawless database to be achieved, user requirements that spell the business rules that a database must comply with are defined. These rules have to be translated into constraints to be used to validate any data submitted to the database before an atomic transaction is executed.

Various types or rules determine the type of constraint that is enforced. In our investigation, we dealt with four granularity levels of constraints, they include: attribute, object, table and database constraints. These constraints affect the database attributes and the spatial geometries. In our research study, we investigated the spatial database consistency in web application framework with Django being our case study. We found out that most web applications maintain integrity controls partly in the application layer and partly in the database layer.

Maintaining, constraints at two layers compromised spatial consistency and therefore we came up with philosophy of having all constraints, that is, implicit and explicit constraints to reside in the database layer. To achieve this, we developed a design method that allowed for the creation of integrity-preserving functions that reside in the database and calling them into the application using a Remote Procedure Call API. The functions were meant to wrap the data and pass data through a series of validation units that are based on data granularity. Granularity based constraint validation means that once a database has passed successfully through database constraints it is considered to be inherently attribute and table consistent as explained in Chapter 2.

Therefore, to have a spatially consistent web application, we advice the use of constraint design method and the prototype design architecture as discussed in Chapter 3. Moreover, a developer should be cautious when using triggers, that is, triggers should be be avoided in the validation process but triggers can be used for data modification to promote spatial database consistency.

5.4 Recommendation

During the research study, new questions arose that could not be answered in the research project due do to hardship and lack of required resources. Hence, we came up with a number of recommendations that need to be investigated further in future work.

5.4.1 Creation of the PL/pgSQL wrapper in the ORM

The current Django ORM has SQL wrapper that generates SQL statements that creates schemas and query the database. However, Django lacks a PL/pgSQL wrapper that would have made the entire process of creation of the database integrity-preserving functions using Python in Django possible. This would have introduced the concept of PL/pgSQL code generation. Therefore, this will enable ordinary web developers to easily and rapidly create complex spatial business logic from Django that ensure web applications to run on spatially consistent database.

5.4.2 PL/Python

PostgreSQL also supports server-side python language called PL/Python. We would recommend a future study on how PL/Python can be used together with Django with regard to creation of functions in the database. Future research study should investigate the advantages of PL/Python compared to PL/pgSQL and what are its limitations. PL/Python may be easier for Django developers who already have skills in python. However, just like PL/pgSQL, a wrapper is still required to allow for code generation. Python being an object-oriented language, we believe that it is better placed than the procedural server-side PL/pgSQL programming language.

5.4.3 Model-driven design

We also recommend the re-engineering of Django to adapt a model-driven approach. Such an approach would support the modeling process when creating web applications from conceptual stage, then logical, and finally the physical web application as a product, ready for deployment. The system developer will be left with the task to define application components, constraints and multiplicities, navigations and database schema as class diagrams using the Object Constraint Language (OCL) and the Web Modeling Language (WebML).

5.4.4 Record of constraints

We suggest the use of a tracking chart that shows the number of validation constraints for a data structure granularity per function. The organization will be in four levels, that is attribute constraints, object constraints, table constraints and database constraints. This will help to create a metadata to locate various constraints for each integrity-preserving function in a database. The metadata record will help in keeping inventory of the constraints performance, for instance, how constraints are violated and how the constraints have been improved when bugs are reported. With this kind of data, the developer will be able to trace the constraints that are most the violated with the aim of improving the system.

5.4.5 Development of Django and GeoDjango installer

Inasmuch as creation of spatial web application using the Django framework is easy and rapid, the learning curve is a bit steep. First, installation and configuration of the system is complicated for someone with no knowledge in programming or Python. This is because a mistake made by putting the applications files outside the PYTHONPATH might result in hidden errors. There, setting, url, model, view and form module use Python classes and methods that need to be imported and called appropriately. We recommend that an installer be developed that will make the installation and configuration of the Django web framework easy and fast for new and less experienced developers. This will make them focus more on programming the web application. The installer should also give database choices, for instance, PostgreSQL and its POSTGIS extension.

5.4.6 To determine the efficiency of taxonomic granularity enforcement of constraints

With a critical look at the taxonomic granularity enforcement in function, there arises weakness when data validation is done in a hierarchical order based on granularity. Despite taxonomic method encompassing all the constraints, there arises the issue of system efficiency. This is because the integrity-preserving function may have raised an exception and stopped earlier had it started with the reference keys which is a database constraint rather than going through attributes then table constraints and stopping at the database constraints. Therefore, if we place the database constraints at the last stage then the system will have to rollback after spending system's resources in validating the object, tuple and table constraints. Although this is subject to testing to determine which method is more efficient, that is, random validation versus the hierarchical taxonomic granularity. We would recommend for a future study to be done to see the performance quality of the taxonomic granularity validation versus the random validation process.

5.4.7 Django for mobile devices

Since most applications are going mobile, I would recommend a Django application that can be operated on the mobile devices that are fitted with GPS devices in future, this will go a long way in helping the development for location-based web application that are spatially consistent. The databases can be located at a central server to serve as the hub for multiple users of the application to enter and get spatial information that are spatially consistent. This will require a middleware that will link the edit map interface with the GPS device to enable the creation of geomtric features and attribute data in real time.

5.4.8 Promote publication on spatial database consistency in web frameworks

The GeoDjango user group is still very small. Also finding literature on geographical information web frameworks is hard. The online user group that focuses on GeoDjango is very small. Scientific papers on the Web GIS database consistency are hard to find. We would recommend more scientific papers and books to be written in this field that are related to web GIS to boost the publicity and interested in this field.

Bibliography

[1] J.D. Anderson. *Cake PHP A Rapid Development Framework*. Cake Software Foundation, version 0.10 edition, 2006.

[2] C. Bauer and G. King. *Hibernate in Action: Practical Object/Relational Mapping*. Manning Publications, 2004.

[3] J. Bennett. *Practical Django Projects*. Apress, 2008.

[4] F. Cagnacci and F. Urbano. Managing wildlife: A spatial information system for GPS collars data. *Environmental Modelling & Software*, 23(7):957–959, 2008.

[5] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. Tools for model-based development of web applications. In *Designing Data-Intensive Web Applications*, pages 499–517. Morgan Kaufmann, San Francisco, 2003.

[6] D. Chak. *Enterprise Rails*. O'Reilly Media, 2008.

[7] T.M. Connolly and C.E. Begg. *Database Systems, A Practical Approach to design, Implementation and Management*. Addison-Wesley, 2005.

[8] L. Daly. *Next Generation Web Frameworks in Python*. O'Reilly, 2007.

[9] R. de By. Designing spatial databases: from reality to a reality. Lecture Notes 2008.

[10] A. U. Defense. *Systems Engineering Fundamentals*. Defense Acquisition University Press, 2005.

[11] K.R. Dittrich, A.M. Kotz, and J.A. Mülle. An event/trigger mechanism to enforce complex consistency constraints in design databases. *ACM SIGMOD Record*, pages 22–36, 1986.

[12] A. Holovaty and J. Kaplan-Moss. *The Definitive Guide to Django Web Development Done Right*. Apress, 2007.

[13] K. Janne and M. Tommi. Partitioning web applications between the server and the client. *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 647–652, 2009.

[14] M. Jazayeri. Some trends in web application development. *IEEE Computer Society Digital Library*, pages 199–213, 2007.

[15] G. Kappel, E. Michlmayr, B. Proll, S. Reich, and W. Retschitzegger. *Web Engineering*. John Wiley & Sons, Ltd, 2006.

[16] B. Kelly, L. Nevile, EA Draffan, and S. Fanou. One world, one web but great diversity. *ACM International Conference Proceeding Series*, Vol. 317:141–147, 2008.

[17] J. Kienzle and S. Gélineau. Ao challenge - implementing the acid properties for transactional objects. *Proceedings of the 5th international conference on Aspect-oriented software development*, pages 202–213, 2006.

[18] S. Mash. *Beginning GeoDjango: Rich GIS Web Applications with Python*. Apress, 2009.

[19] D. Moore, R. Budd, and W. Wright. *Professional Python Frameworks Web 2.0 programming with Django and Turbogears*. Wrox Press Ltd, 2008.

[20] J. Musser, T. O'Reilly, et al. *Web 2.0 Principles and Best Practices*. OReilly, 2006.

[21] E.J. O'Neil. Object/relational mapping 2008: Hibernate and the Entity Data Model (EDM). *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1351–1356, 2008.

[22] E. Oren, A. Haller, M. Hauswirth, B. Heitmann, S. Decker, and C. Mesnage. A flexible integration framework for semantic web 2.0 applications. *Software, IEEE*, 24(5):64–71, 2007.

[23] N. Paladi. Model based testing of data constraints. *Proceedings of the 8th ACM SIGPLAN workshop on Erlang*, pages 71–82, 2009.

[24] J. K. Peeter. Database management. http://pkirs.utep.edu/, 2004.

[25] C.J. Pilgrim. Improving the usability of web 2.0 applications. *Proceedings of the nineteenth ACM conference on Hypertext and hypermedia*, pages 239–240, 2008.

[26] F. Pinet, M. Duboisset, and V. Soulignac. Using UML and OCL to maintain the consistency of spatial data in environmental information systems. *Elsevier Science Direct*, Volume 22:1217–1220, 2006.

[27] V. Ramachandran. Design patterns for building flexible and maintainable j2ee applications. http://java.sun.com/developer/technicalArticles/J2EE/despat/, January 2002.

[28] M.M. Reek. A top-down approaach to teaching programming. *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, pages 6 – 9, 1995.

[29] C. Richardson. ORM in dynamic languages. *May/June 2008 ACM QUEUE*, 6 Issue 3:30–37, 2008.

[30] W. Scacchi. Process models in software engineering. *Encyclopedia of Software Engineering*, 2, 2001.

[31] V. Thakur. *ASP.NET 3.5 Application Architecture and Design Build robust, scalable ASP.NET applications quickly and easily*. Packt Publishing, 2008.

[32] P. Thiran, J.L. Hainaut, G.J. Houben, and D. Benslimane. Wrapper-based evolution of legacy information system. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol 15 , Issue 4:329 – 359, 2006.

[33] P. Van Zyl, D.G. Kourie, and A. Boake. Comparing the performance of object databases and ORM tools. page 11, 2006.

[34] M. Veit and S. Herrmann. Model-view-controller and object teams: A perfect match of paradigms. *140 - 149*, pages 140 – 149, 2003.

Appendix A

Settings.py

```
# Django settings for brazil project.
import os
baseDirectory = os.path.dirname(__file__)
fillPath = lambda x: os.path.join(baseDirectory, x)
staticPath, templatePath = map(fillPath, ['static', 'templates'])
DEBUG = True
TEMPLATE_DEBUG = DEBUG
ADMINS = (
# ('Your Name', 'your_email@domain.com'), ) MANAGERS = ADMINS
DATABASE_ENGINE = 'postgresql_psycopg2'
DATABASE_NAME = 'prototype' # this is spatial database not an ordinary one
DATABASE_USER = 'postgres'
DATABASE_PASSWORD = 'banana' # Not used with sqlite3.
DATABASE_HOST = ' ' # Set to empty string for localhost .
DATABASE_PORT = ' ' # Set to empty string for default.
# Local time zone for this installation. Choices can be found here:
# http://en.wikipedia.org/wiki/List_of_tz_zones_by_name
# although not all choices may be available on all operating systems.
# If running in a Windows environment this must be set to the same as your
# system time zone.
TIME_ZONE = 'America/Chicago'
# Language code for this installation. All choices can be found here:
# http://www.i18nguy.com/unicode/language-identifiers.html
LANGUAGE_CODE = 'en-us'
SITE_ID = 1
# If you set this to False, Django will make some optimizations so as not
# to load the internationalization machinery.
USE_I18N = True
# Absolute path to the directory that holds media.
# Example: '/home/media/media.lawrence.com/'
MEDIA_ROOT = staticPath
# URL that handles the media served from MEDIA_ROOT. Make sure to use a
# trailing slash if there is a path component (optional in other cases).
# Examples: 'http://media.lawrence.com', 'http://example.com/media'
```

```
MEDIA_URL = '/static/'
# URL prefix for admin media – CSS,JavaScript and images.
# Make sure to use atrailing slash.
# Examples: 'http://foo.com/media/', '/media/'.
ADMIN_MEDIA_PREFIX = '/media/'
# Make this unique, and don't share it with anybody.
SECRET_KEY = 'iˆoow(aquo(&&u7&hd5+1hhm88*x94c-8p=xe(l4=_%7l ˆ$xmf'
# List of callables that know how to import templates from various sources.
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.load_template_source',
    'django.template.loaders.app_directories.load_template_source',
    # 'django.template.loaders.eggs.load_template_source', )
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware', )
ROOT_URLCONF = 'brazil.urls'
TEMPLATE_DIRS = (
    #'C:/amazonia/brazil/template',
    templatePath,)
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',
    'django.contrib.gis',
    'brazil.avian',
    'olwidget',)
```

Appendix B

Model.py

```
from django.contrib.gis.db import models
class SpatialRefSys(models.Model):

class Amazonialegal(models.Model):
        id = models.IntegerField(primary_key=True)
        amazonialegalid = models.IntegerField()
        buffer = models.MultiPolygonField()
        geom = models.MultiPolygonField()
        objects = models.GeoManager()
        class Meta:
                db_table = u'amazonialegal'


classInterfluves(models.Model):
       fid = models.IntegerField()
       fluvename = models.CharField(max_length=80)
       geom = models.MultiPolygonField()
       objects = models.GeoManager()
       class Meta:
              db_table= u'interfluves'


class Distrib(models.Model):
        gid = models.IntegerField(primary_key=True)
        cidx = models.DecimalField(max_digits=20, decimal_places=0)
        distribgeom = models.MultiPolygonField()
        objects = models.GeoManager()
        class Meta:
                db_table = u'distrib'


class Speciesdistrib(models.Model):
        speciesid = models.IntegerField()
        interfluveid = models.IntegerField()
        class Meta:
```

```
        db_table = u'speciesdistrib'


class Family(models.Model):
    fidx = models.IntegerField(primary_key=True)
    famname = models.CharField(max_length=50)
    taxorder = models.IntegerField(unique=True)
    class Meta:
        db_table = u'family'


class Genus(models.Model):
    gidx = models.IntegerField(primary_key=True)
    genusname = models.CharField(max_length=50)
    taxorder = models.IntegerField(unique=True)
    familyid = models.ForeignKey(family, db_column='fidx')
    class Meta:
        db_table = u'genus'


class Species(models.Model):
    cidx = models.IntegerField(primary_key=True)
    authority = models.CharField(max_length=50)
    eng_name = models.CharField(max_length=50)
    listname = models.CharField(max_length=50)
    port_name = models.CharField(max_length=50)
    taxorder = models.IntegerField(unique=True)
    genus = models.ForeignKey(genus, db_column='gidx')
    class Meta:
        db_table = u'species'


class Call_Dbase_famfunctions():
    def call_family(self, familyid, famname, taxorder):
        cursor = connection.cursor()
        family = cursor.callproc('add2_family', (familyid, famname, taxorder))
        cursor.execute('COMMIT')
        cursor.close()
        return family


class Call_Dbase_genfunctions():
    def call_genus(self, genusid, genusname, taxorder, familyid):
        cursor = connection.cursor()
        genus = cursor.callproc('add2_genus', (genusid, genusname, taxorder, familyid))
        cursor.execute('COMMIT')
        cursor.close()
        genus
```

```
class Call_Dbase_spfunctions():
    def call_species(self, speciesid, authority, eng_name, listname,
        port_name, taxorder, genus):
        cursor = connection.cursor()
        species = cursor.callproc('add2_species', (speciesid, authority,
                eng_name, listname, port_name, taxorder, genus))
        cursor.execute('COMMIT')
        cursor.close()
        return ret


class Call_Dbase_distfunctions():
    def call_distrib(self, speciesid, speciesgeom):
        cursor = connection.cursor()
        distrib = cursor.callproc('add2_distrib', (speciesid, speciesgeom))
        cursor.execute('COMMIT')
        cursor.close()
        return distrib
```

Appendix C

Form.py

```
from django import forms
from django.contrib.gis.db import models
from django.contrib.gis import forms
class CreateForm(forms.Form):

class FamilyForm(forms.Form):
        fidx =forms.IntegerField()
        famname =forms.CharField(max_length=50)
        taxorder =forms.IntegerField()


class GenusForm(forms.Form):
        gidx =forms.IntegerField()
        genusname =forms.CharField(max_length=50)
        taxorder =forms.IntegerField()
        familyid =forms.IntegerField()


class SpeciesForm(forms.Form):
        cidx =forms.IntegerField()
        authority =forms.CharField(max_length=50)
        eng_name =forms.CharField(max_length=50)
        listname =forms.CharField(max_length=50)
        port_name =forms.CharField(max_length=50)
        taxorder =forms.IntegerField()
        genus =forms.IntegerField()


    # Distrib
class DistribForm(forms.Form):
    distributionid =forms.CharField(max_length=100)
    distribgeom = forms.CharField(widget=EditableMap(options=
    'name': 'Avian Distribution','layers': [ 'google.hybrid', 'google.streets',
    'google.physical','google.satellite',], 'geometry': 'polygon', 'is_collection':
    True,'default_lat': -8, 'default_lon': -50, 'editable': True, 'hide_textarea':
    False,'map_div_style': 'width': '600%', 'height': '550px',  ))
```

Appendix D

View.py

```
from django.shortcuts import get_object_or_404, render_to_response
from django.http import HttpResponseRedirect, HttpResponse
from django.core.urlresolvers import reverse
from django.core.exceptions import *
from django.template import RequestContext
from brazil.avian.models import *
from django.contrib.gis.db import models
from django.contrib.gis import admin
from olwidget.widgets import EditableMap
from olwidget.widgets import MapDisplay
from olwidget.widgets import InfoMap
from django.avian.forms import *
from django.template.loader import get_template
from django.template import Context
from django.http import HttpRespon import datetime


class def family(request ):
        if request.method == 'POST':
           form= FamilyForm(request.POST)
        if form.is_valid():
           my_funct = Call_Dbase_famfunctions()
           family = my_funct.call_family(request.POST['fidx'],
           request.POST['famname'],request.POST['taxorder'])
           return HttpResponseRedirect('/family/')
        else:
              form = FamilyForm()
        return render_to_response('family.html',  'form':form )


def genus(request ):
     if request.method == 'POST':
         form = GenusForm(request.POST)
     if form.is_valid():
```

```
        my_funct = Call_Dbase_genfunctions()
        family my_funct.call_genus(request.POST['gidx'],request.POST['genusname'],
        request.POST['taxorder'],request.POST['familyid'])
        return HttpResponseRedirect('/genus/')
    else:
        form = GenusForm()
    return render_to_response('genus.html', 'form':form , 'error': True ,)


def species(request ):
    if request.method == 'POST':
        form = SpeciesForm(request.POST)
    if form.is_valid():
        my_funct = Call_Dbase_spfunctions()
        species = my_funct.call_species(request.POST['cidx'],
        request.POST['authority'],request.POST['eng_name'],
        request .POST['listname'],request.POST['port_name'],
        request.POST['taxorder'],request.POST['genus'])
        return HttpResponseRedirect('/species/')
    else:
        form = SpeciesForm()
    return render_to_response('species.html', 'form':form , 'error': True ,)


def distrib(request ):
    if request.method == 'POST':
        form = DistribForm(request.POST)
    if form.is_valid():
        my_funct = Call_Dbase_distfunctions()
        distrib = my_funct.call_distrib(request.POST['distributionid'],
        request.POST['distribgeom'])
        return HttpResponseRedirect('/distrib/')
    else:
        form = DistribForm()
    return render_to_response('form.html', 'form':form , 'error': True ,)


def interfluves(request):
    fields = [instance.distribgeom for instance in Distrib.objects.all()]
    map = MapDisplay(fields=fields)
    returnrender_to_response('map.html', 'map': map)


    #amazonialegal
def amazonialegal(request):
    instance = Amazonialegal.objects.all()[0]
    map = MapDisplay(fields=[instance.geom],options='
    layers':['google.hybrid',
```

```
    'google.streets','google.physical',
    'google.satellite',], 'geometry': 'polygon', 'is_collection':
    True , 'default_lat': -8,'default_lon': -50, 'hide_textarea':
    False ,'map_div_style': 'width': '600px', 'height': '550px', , )
return render_to_response('amazonialegal.html', 'map': map)

    #infocursor
def interfluveinfo(request):
    instance = Distrib.objects.all()[10]
    map = InfoMap([[instance.distribgeom,
    'html': instance.gid,
    'style': 'fill_color': ' #03FF02',],]) return r̄ender_to_response('map.html', 'map':
map)

defmapping(request):
    fields = [instance.geom for instance in Avianite.objects.all()]
    map = MapDisplay(fields=fields)
    return render_to_response('map.html', 'map': map)
```

Appendix E

Url.py

```
from django.conf.urls.defaults import *
from django.contrib.gis import admin
from brazil.avian.views import mapping, interfluves,info,
    interfluveinfo, home , renderdistrib, visualization , family
from brazil.avian.views import amazonialegal,genus, species,
    distrib, amazonhome, amazonvisualization #,message
from brazil.avian.views import message
# Import custom modules
import settings
admin.autodiscover()

    urlpatterns = patterns(' ',
    (r'^admin/', include(admin.site.urls)),
    (r'^test/', message ),
    (r'family/' , family),
    (r'genus/' , genus),
    (r'species/' , species),
    (r'distrib/' , distrib),
    (r'amazonvisual/' , amazonvisualization),
    (r'amazonhome/' , amazonhome),
    (r'amazonlegal/' , amazonialegal),)
    if settings.DEBUG:
# Set
mediaURL = settings.MEDIA_URL[1:]
# Extend
urlpatterns += patterns(' ',
(r' ^% s(?P <path >.)$' % mediaURL, 'django.views.static.serve',
'document_root': settings.MEDIA_ROOT),)
```

Appendix F

Template

```
<head >
    {{ form.media }} {{ map.media }}
    <table border=0 cellspacing=0 cellpadding=5 width=100% class='bg_nav_left_header'>
    <tr >
    <td ><a class='nav_left_header_text' href='http://127.0.0.1:8000/home/' title='FrontPag'
    ><h1 >Home </h1 ></a >  </td >
    </tr >
    </table >
    </head >
    <body >
    {% if form.errors %}
    <pstyle='color: red;' >
    Please correct the error{{ form.errors—pluralize }} below.
    </p >
    {% endif %}
    <h1 >Amazonian Web GIS Application </h1 >
    <form action='.' method='POST' >
    <table >
    form  map
    </table >
    <p ><input type='submit' value='Submit' class='actiontable' ></p >
    </form >
    </body >
    <head ></head >
    <body >{{ map }} </body >
```