A SPATIALLY-EXPLICIT DYNAMIC DATA MANAGEMENT FACILITY TO SUPPORT LOCATION-BASED CLIMATE SERVICES

FARZIN ASHOURI
March, 2013

SUPERVISORS:

Dr.Ir. R.A. de By
Dr. J.M. Morales

A SPATIALLY-EXPLICIT DYNAMIC DATA MANAGEMENT FACILITY TO SUPPORT LOCATION-BASED CLIMATE SERVICES

FARZIN ASHOURI
Enschede, The Netherlands, March, 2013

Thesis submitted to the Faculty of Geo-information Science and Earth Observation of the University of Twente in partial fulfilment of the requirements for the degree of Master of Science in Geo-information Science and Earth Observation.
Specialization: GFM

SUPERVISORS:

Dr.Ir. R.A. de By
Dr. J.M. Morales

THESIS ASSESSMENT BOARD:

Dr. R. Zurita-Milla (chair)
V. de Graaff MSc

ABSTRACT

Availability of climatic data from a variety of sources and meteorological organizations does not guarantee its accessibility and usability for the end user. The available data is only usable for the experts who are in the geoinformation or meteorology profession. On the other hand, timely climatic information and analysis are in demand for many professional communities. Among those, farmers benefit most from this information; because agricultural products are exposed to and are affected by weather conditions, and the climatic record of a farm has an impact on its products.

Efforts to build climatic databases have led to creation of geoportals that are mostly fed from meteorological stations. To approximate the points in between the stations they should be dense enough, and this is not the case at least in most developing countries. There are some data sources that provide climatic data from remote sensing satellites. The provided data are continuous raster data, even though in some cases the number of missing pixels is considerable. In this research project, we are building a continuously updated climatic database, based on data that is acquired from meteorological and environmental satellites for a number of climatic parameters. Even though the resolution of the data measured with satellites is limited and their accuracy is dependent upon atmospheric conditions, construction of such a database is useful since it delivers data for every location in the study area.

The problem of missing pixels can be solved by aggregation methods. Besides, summarized information over a period of time is needed and can also be achieved by aggregation techniques. Due to the need for extraction of summarized information from the database, a number of aggregation techniques have been facilitated as a part of the system. Construction of the database and its continuous updates are fully automatic, and aggregation functions result in the required answers based on the input parameters. The framework is a basis for a location-based climate web service.

Keywords

spatial databases, temporal aggregation, climate services, dynamic data management

ACKNOWLEDGEMENTS

I am truly indebted to my first supervisor, Rolf de By for his continuous support and guidance. His defining characteristic and high scientific standards set an example for me.

I would like to express my gratitude to my second supervisor, Javier Morales for helping me in technical issues. Furthermore, I have been privileged to meet a professional scientific programmer, Bas Retsios who help me in implementation part this research project. I would like to thank Clarisse Kagoyire who shared her experience in her PhD research project with me. I am also grateful to my programming instructor, Farsheed who has given me a deeper insight into programming. I acknowledge my special gratitude to Ali Abkar for his supportive role during this course. I want to thank my friends, Ali and Manuel who played an important part in conducting a better research.

Last but not least, I would like to thank my supportive family and specially my caring mother for her constructive role in my life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction

1.1 MOTIVATION AND PROBLEM STATEMENT

With recent developments in spatio-temporal data acquisition methods, data is available for researchers to study different aspects of climate change. Availability of data in both spatial and temporal dimensions, makes the data management issue more critical. Near real-time climatic data from a variety of sources are updated continuously. There are a number of challenges regarding climatic data management. One of them is acquiring the climatic data for a specific area and specific moment in time that satisfies user demand in terms of quality of the data and temporal and spatial resolution. Another challenge is putting all the processed information together in a spatio-temporal warehouse for further analysis.

The user may need climatic information that is updated on a daily basis. There are some data sources that provide such data. One of the technically challenging issues is near real-time populating of the climatic spatio-temporal database as the data in the data source is updated. Because of the large amount of data generated by the typical data stream, it is not viable to store the data in a way that is ready for answering queries [35]. That is the reason most applications just adopt aggregate queries on data streams. What is important for researchers, decision-makers or professional communities that intend to use such a system, is information at the desired aggregation level with different spatial and temporal granularities.

Timely weather condition data and weather statistics, are highly in demand in a variety of applications. Productivity in agriculture is highly dependent upon favorable weather conditions. Human beings (almost) do not have any control over climate. One the other hand, every climatic situation, calls for certain measures or counter-measures.

Potential impacts of existing weather conditions on productivity of the farm crops can be assessed with regard to favorable conditions at a particular time of the year. It is particularly important for farmers to take appropriate measures when they observe important deviations of current weather from normal or favorable situations. Moreover, the impact of climate variability and climate change on productivity of agricultural products can be studied. Data from meteorological organizations can also be an input for the database. As a result, the system can warn farmers of upcoming weather conditions more efficiently. At the moment, the focus is on automatic importing of existing weather parameters, but the system will have the capability of manual importing of forecast data.

This can also be a foundation for policy-makers to take the appropriate measures. In this research project, we are developing a framework for updated climatic data and weather statistics for coffee farms in Rwanda. This framework can also be used for on-farm decision-making by farmers and for taking counter-measures.

The proposed system is open source and fully documented. This helps developers to alter the framework according to their dataset and study area and also their requirements. Accordingly, the framework can be used for studying climate in other parts of the world. During this research project, we will focus on our case study, but the prospective system will have the capability to be

adapted to other studies of this kind.

It is worth to mention that in this research project, the idea is to develop the framework to inform farmers about current weather conditions and also weather statistics. Therefore, research on the areas of favorable conditions for crops and decision making is beyond the scope this study. Another important issue is that despite the fact that facilitating the weather statistics according to the existing trends is a part of this study, we do not claim that this framework has the capability to adopt complicated weather forecast models to predict weather conditions.

1.2 RESEARCH IDENTIFICATION

1.2.1 Research objectives

Throughout this research project, we aimed at construction of a framework to acquire, manipulate, and absorb continuously updated climatic and environmental data as a dynamic spatio-temporal database. To this end, a number of other steps should be followed.

- **Objective** Building a continuously updated spatio-temporal database for climate data, for coffee farming in Rwanda.
- **Subobjective 1.** Making a prototype system for a few data products to make sure that all the processes is feasible.
- **Subobjective 2.** Making a list of parameters and data products that are most suitable for the system.
- **Subobjective 3.** Designing a database that can accommodate such a dynamic system.
- **Subobjective 4.** Facilitating aggregate queries over data streams and also designing a data aging mechanism.
- **Subobjective 5.** Defining a function to calculate weather statistics that results in an array, based on input parameters, from which time evolving arrays can be drawn. Based on the characteristics of the climatic parameter, *sum* or *average* aggregate functions can be used to extract summarized information, and some others like *variance* or *standard deviation* to show the variability.

1.2.2 Research questions

All research questions will be answered in the context of accomplishment of the main objective.

1. How to design a database to support continuous climatic data uptake and aggregation. Temporal and spatial aspects of the data as well as nature of climatic data should be considered in the design.
2. How to design an application that can operate around-the-clock to import new updated data into the database.
3. What is the need for summarized information and statistical analysis and what kind of questions may arise for research community, farmers and decision-makers that is retrievable from this database?
4. What is the most suitable data structure in the database? Data structure can be raster or vector.

5. What are the most suitable data products?
6. How to manage large amounts of data streams?
7. How to define a weather statistics function? Inputs and outputs of the functions should be selected carefully

1.2.3 Innovation aimed at

This study consists of a number of phases, namely, acquiring climatic data for a specific region, continuous populating of a spatio-temporal database and performing some statistical analysis as a framework to inform farmers of weather conditions. All these phases are done separately and we follow the existing methods to develop our own system. This provides a framework to compare the existing weather conditions with favorable conditions for coffee farming.

Needless to say, among existing methods we are looking for the most suitable one to adopt. Besides, our proposed framework is open-source and fully documented. So, the users who want to make use of other datasets even for another study area can use the system with minor changes to be fitted with their dataset.

1.3 PROJECT SET-UP

This research project has various technical and computational parts, all of which are important to make the system work.

1.3.1 Plan of the project and methods adopted

In this section, detailed steps one should follow to build a fully functional system are explained.

New data detection mechanism

There are a number of data sources that are updated on a regular basis and we are going to use them throughout this research as the data source. For the system to be fully automatic, it is important that the system automatically recognizes the newly added data in the data source.

Data selection and data acquisition

What kinds of information need to be put in the database including the parameters, data products and study area should be identified before starting to build the system. A number of climatic parameters should be selected. In the selection, a couple of issues should be taken into consideration. First, it should be tried to choose the data products that issue updates on regular basis and their updates are publicly available through a network or server. The other criteria for selection is the continuous nature of it. This means that, for every point in the study area, the data should be available. Raster data provided by environmental and meteorological satellites provides such information. The other option is using interpolated data from meteorological stations. Since the target for this study is developing countries and such stations are not dense enough for interpolation, we opt out of this option.

Metadata extraction

There should be a way to extract some information about the product. Projection system, coordinate system, number of rows and columns, pixel size, corner coordinates, number of bands

and format are some of metadata to be extracted. It is not part of an automatic process, but it is important in building the system.

Preprocessing

The files should undergo a number of preprocessing steps before they can be entered into a database. For example, some of the raster products that are used in this project are not in a format that can be readily imported to the database. So, a format conversion is needed for them, so that the database can be built up based on data that is converted.

The data inputs are in raster format and normally the data have georeference information and just a simple transformation and reprojection is needed to the desired coordinate system and projection system. Reprojection can be helpful when we want to change the projection system of the raster data to other projection system that is considered more appropriate. This operation may involve some errors. But still there is another option that skips this operation and keeps the products in its original spatial reference system. Then just in case the points that user wants to extract from the system are not in the same coordinate system as that of original data set, a transformation is needed prior to preforming the query. There is a discussion for this choice in chapter 6 of the thesis.

The data source we have includes an entire world in most of the cases and it is much better to crop the data, considering the study area. The volume of data will be much lower after cropping. This will be done by comparing the data set with borders of the study area, provided that the spatial reference system of the data set and the border is the same. This can be done either before populating the database or after entering data into the database. If we choose to crop before importing the data into the database, smaller data would be imported into the database, but extra files for the extent of the study area should be somewhere in the compute. If we store the study area extent as a geometry in the database, the cropping operation can be done inside the database. We chose to do it before entering the database, so that the volume of data in the database and also processing inside the database is reduced significantly.

Importing the data in the database

After preprocessing, data should be entered in a database for further analysis. For proper or optimal design, a number of issues should be considered. First issue is that it is important to import all the information stored as raster files in a database. Besides, data volume inside the database should also be taken into consideration. Large amount of data inside the database can also affect the overall system performance that should be avoided and be taken into consideration in the design.

One option is to import the entire raster file in the database, so that we will not lose any information in the study area. It is important to notice that raster file in the database may not be the best way to store the data. It is because of the accumulation of raster files over long periods and volume of the data. The problem of data volume can be solved by aggregation techniques that we will mention later in this document, but this has to be tested. Another issue to be considered is that the system may not be fast enough with raster design.

It is needed to design a database to accommodate relevant information without storing the whole raster file. The other option is to store every pixel as a record with all the information of an entire period. The information could be monthly, daily and eight days aggregated values for example. For testing purposes, I need to populate the database (manually or artificially for now), so that further analysis and temporal aggregation is possible. There is an extensive discussion for this choice in Chapter 6.

Data aging mechanism

Since the data is being built up on a regular basis in the database, obviously the problem of enormous amount of data arises. On the other hand, finer granularity data may not be of interest of the user for the data that is old enough. A data aging mechanism based on this theory will be discussed in Chapter 5.

Extracting information from the database and time-evolving charts

The prospective system will be a basis for a location-based web service, that is capable of informing farmers about their own plots. Everything we know about the plots, will be displayed upon user demand in the form of (time-evolving) charts. Possible information that they may want to find answer from the system is as follows:

- Daily temperature high over last week, this season, last season, or all seasons
- Daily temperature low over last week, this season, last season, or all seasons
- Daily rainfall over last week, this season, last season, or all seasons
- Weekly maximum, minimum or average of the parameters for the current week and corresponding week over multiple years
- A measure of variability such as variance or standard deviation for the aggregated data

1.3.2 Risks and contingencies

The data sources and their updates are usually in networks or servers. These networks or servers are subject to change every time. The automated system that performs the data acquisition on a regular basis, depends on the predefined settings and architecture of these sources. The change in the settings would lead to problems in the automated system. The system should be adaptable with the changes with minor refinement of the system.

1.4 RESOURCES REQUIRED

1.4.1 Data

Possible sources of data are ITC network and freely available sources that are mentioned in section 1.3.1. The exact location of coffee farms are needed to assign climate data to them.

1.4.2 Software and hardware

Postgresql, PostGIS, GDAL and Python programming language are the software requirements. Personal computer and having remote access to ITC server is also important for this project.

Chapter 2
Literature review

2.1 INTRODUCTION

In recent years, the demand for comprehensive and timely global agriculture intelligence in the form of digital spatial climatic data sets has increased considerably. Securing reliable and stable supply of food, calls for the timely information on crop production. Minimum and maximum temperature and precipitation are the basic climate elements that are provided by climatic data sets. Despite the great importance of climate data inputs and availability of them, many users do not have the appropriate background understanding of extracting relevant information from climatic data sets.

Nowadays, very fine resolution climate grids (1 km) are available from a variety of sources. Typical distance of meteorological stations is in the order of 100 km, except for populated areas in developed countries [11]. Coarser data (50 km) was generated using computers in the past. In this study, we are looking at some climate parameters that are available all over the study area. In this chapter, we want to explore the state-of-the-art in the field of climatic databases and the techniques to make use of them properly.

2.2 CLIMATE DATABASES

2.2.1 Climate databases for general purposes

National Climatic Data Center (NCDC) is the world largest geoportal of climate and weather data (www.ncdc.noaa.gov). It provides climate and weather data and publishes this for global use. Hourly global data just for the discrete stations all over the world are existed. They are actively produce data for a number of climatic parameters like temperature and precipitation. Different organizations and sources such as *World Meteorological Organization (WMO)* provide information for NCDC.

There are some time periods that are missing from these databases and they are available in the inventory file of the geoportal. The data is available from 1973 onwards, and required information can be ordered to be sent to an e-mail address (www.climate.gov). The main problem with this service, that makes it inappropriate, is that it offers climatic information in discrete stations. Besides, for most of the developing world the stations are not dense enough to allow for interpolation for the points in between. For example, in our study area Rwanda, just a few stations throughout the country are available. As long as these stations provide *in situ* data, it is expected that they are more reliable than remotely sensed data. So, they can be used for quality control in the system.

The *Global Historical Climatology Network (GHCN)* provides daily data and it is also available via NCDC like WMO stations. The stations are more sparse than that of WMO. For the study area there are no GHCN stations available. It mostly holds historical data.

A high-quality 103 year data set of monthly maximum and minimum temperature and precipitation on a four km grid over the conterminous US has become accessible in 2002 [18].

Global Precipitation Climatology Center (GPCC) has been founded in 1989 on request of WMO. Gridded, gauge-based monthly precipitation products are available in $1.0° \times 1.0°$ and $2.5° \times 2.5°$ geographical latitude by longitude system [46]. Better spatial resolution as much as $0.25° \times 0.25°$ is provided by GPCC's new global climatology and is available for download via (gpcc.dwd.de). The first version of combined precipitation dataset of the *Global Precipitation Climatology Project (GPCP)* covers the period of 1987 through 1995. It contains monthly global precipitation and is an integrated precipitation analysis estimates from *low-orbit satellite Infra Red (IR)* data [28]. The version 2 of the product is available from 1979 to present [2].

Hijmans et al. [26] described the development of a database of monthly climatic data from various sources for precipitation and maximum, minimum and mean temperature which is restricted to the period of 1950–2000. Mitchell and Jones [38] explained the construction of a database of global monthly climate observation from meteorological stations that has six climate parameters. Global coverage (excluding Antarctica) is achieved by interpolating monthly climate observations from meteorological stations into $0.5°$ grid. The data set is called *CRU TS 2.1* and is publicly available via (www.cru.uea.ac.uk).

2.2.2 Climate databases for agriculture

Hertel et al. [25] proposed an infrastructure for researchers working in the area of agriculture, land use and the environment to study sustainability of global agricultural systems in the long run.

The *GAEZ (Global Agro-Ecological Zones)* model and its associated database is an important attempt to build spatially-explicit global data for long-term environmental agricultural analysis [25]. It is a joint effort by the *Food and Agriculture Organization of the United Nations (FAO)* and the *International Institute for Applied Systems Analysis (IIASA)*. The *Global Agriculture Monitoring Project (GLAM)* is a joint USDA, NASA, SDSU and UMD initiative to construct a global agriculture monitoring system [6]. Its focus mainly is on vegetation index time series, near-real time surface reflectance and value-added products like cropland masks.

2.3 SPATIAL AND TEMPORAL AGGREGATION

Aggregate functions are popular in database applications. The popularity stems from their ability to provide summarized information from large amounts of data. Fundamentals and definitions of aggregation have been studied by Smith and Smith [50]. Gray et al. [19] thoroughly studied classification of aggregation functions. They are widely used in applications such as *On Line Analytical Processing (OLAP)*, statistical evaluation, decision support and spatial data management [9, 7, 19, 24, 55].

Lopez et al. [35] conducted a comprehensive survey on spatio-temporal aggregate computation. The most suitable techniques for evaluation of aggregate queries on spatial, temporal and spatio-temporal data were studied in the research. The same study also proposed a model that makes it possible to compare and analyze different existing techniques for the evaluation of aggregate queries. Klug [33] precisely defined aggregate functions and provided a framework for defining aggregate functions for relational databases.

Temporal aggregation is the process of time partitioning and grouping the tuples over these time partitions. These groups are called *granules*. To form a coarser time granularity, the granules in one time granularity are further aggregated [35]. OLAP queries facilitate aggregation across a number of columns in a relation [24].

View materialization is widely in practice in databases and data warehouses and other analytical environments to reduce the load of the operational database. A comprehensive study about

views, their advantages and their inherent deficiencies was conducted by Halevy [22]. Aggregated views can be used to accelerate query response time in analytical environments [20].

2.4 STREAMING DATA AND AGGREGATION OVER DATA STREAMS

2.4.1 Data streams

Ordered sequence of value points that are received/read in increasing order are called *data streams* [4]. Since a data stream is always associated with timestamps, data streams can be considered as a special case of temporal data. It is extremely expensive to store the streaming data in such a way that is readily available for answering queries. This is because of the enormous amount of data that is generated by data streams. Most applications tend to perform aggregate queries over data streams to get summarized information, and store this instead of original data. In data streams summarized information is often more important than specific data entries themselves [44].

2.4.2 Aggregation over data streams and continuous queries

The sliding window model and complete model are two main models for processing streaming data [12, 56]. Babu and Widom [5] extensively analyze the problem of query processing over data streams. Definition and evaluation of continuous queries over data streams, semantic issues as well as efficiency concerns are studied in the research. When only recent values of the data are of interest, the sliding window model is used, while a complete model is for all the values in the stream.

Zhang et al. [56] also proposed a hybrid model called *Hierarchical Temporal Aggregation (HTA)* to address the deficiencies in the two models. In the hybrid model they aggregate earlier data at coarser granularities, but they keep full information of the most recent time. There are two mechanisms to control the model, namely, the fixed storage model and the fixed window model.

Time evolving data in temporal databases or data warehouses needs to be maintained using costly operations of temporal and spatio-temporal aggregation. Zhang et al. [56] examined the problem of performing such aggregates over data streams which are maintained using multiple levels of temporal granularity in which more recent data is aggregated with finer granularities and older data is aggregated using coarser details. Hornsby and Egenhofer [27] presented temporal zooming for spatio-temporal knowledge representation. The study tries to describe how new operations can help to support shifts in levels of detail over time.

A single granularity in the database is insufficient for many real-world applications and semantics of granularities should be embedded in the database design. To deal with this issue, Khatri et al. [32] proposed a spatio-temporal conceptual model for semantics of spatial and temporal granularities.

Spatial, temporal and spatio-temporal aggregates over streams of remotely sensed data using the spatial extent of the raster image data are investigated in [57]. In the study an indexing scheme based on Box-Aggregation Tree was presented to compute spatio-temporal aggregates over streams of imagery that vary in size and position. Computation of basic aggregation functions, such as *average, count, minimum, maximum* and *add* over a multidimensional raster image database is studied in Gutiérrez and Baumann [21]. To achieve better performance, they applied a pre-aggregation framework.

2.4.3 Data stream management systems

There are a number of *Data Stream Management Systems (DSMS)* that following their exact routines is not in our agenda in this study, but we are inspired by some of them. They mostly deal with

the issue of continuous queries which are not the main concern in this study but it may well be the subject of future studies.

TelegraphCQ is a dataflow system that was developed at Berkeley and is used for processing continuous queries over data streams. Volatile data stream environments need an adaptive architecture for supporting dynamic query workloads. There is an implementation of TelegraphCQ using the code base of PostgreSQL [8]. The main idea behind the implementation of this system is sharing and adaptivity [47]. *Related Histogram (RHist)* is an appropriate summarization for data streams [44]. A workload decay model is introduced to ensure that recent query patterns weighted more than older ones.

Mokbel et al. [40] presented a continuous query processor designed for highly dynamic environments such as location aware environments. They implement it inside the *Pervasive Location-Aware Computing Environment (PLACE)* which is a scalable location-aware database server.

Aurora is an architecture and model for data stream management and monitoring applications [1]. Monitoring applications differ from conventional data processing substantially in that continual inputs from different sources should be processed and reacted to. *The Scalable On-line Execution (SOLE)* algorithm was introduced for concurrent continuous spatio-temporal queries over data streams [39].

Arasu et al. [3] focused on defining precise semantics of continuous queries over data streams. The semantics are implemented in the relational query language and window specifications of SQL-99 to map from streams to relations. This leads to proposal and implementation of a robust *Continuous Query Language (CQL)* in a data stream management system. Kazemitabar et al. [31] tried to describe spatial libraries of *Microsoft SQL Server StreamInsight* for geospatial streaming applications. It is an infrastructure to run continuous queries over high-rate data streams.

2.4.4 Data aging

With the emergence of data streams and continuously growing large amounts of data, new challenges arose for effective management of aging data. A mechanism called persistent views was offered for flexible reduction of data volume [48]. Low interest, detailed data can be represented by aggregated data for instance. The study offers a foundation for implementation of persistent views as well. Big, unstructured and streaming in real time data that should be unified in a management system, reminds one of the new challenges ahead of the developer community. Meijer [37] introduces *Language-Integrated Query (LINQ)* as a compelling foundation for big data. It tries to bridge the gap between the world of objects and data by integrating programming languages and databases, using theoretical concepts like monads that allow to abstract from the intricacies of data containers, while presenting useful iterators over those containers.

Chapter 3
Data Sources and Tools

Throughout this chapter, the choice of data sources is scrutinized. In addition, the tools and software packages that are used for this research project will be discussed.

3.1 DATA SOURCES

To build our system, some data sources should be chosen. Data sources should be continuously updated, so that information from every point in the study area is retrievable. To do this, raster data is more appropriate as data source. It can be a satellite product or result of interpolation of *in situ* data. In this project, we are not interested in raw satellite data that should still undergo a long statistical process to produce a final data product. What we are interested in, is a final product like Land Surface Temperature (LST) or precipitation. Despite the fact that we try to choose the most suitable and relatively complete climatic parameters, it is worth to mention that this is not the primary goal of this study, and more useful data products which existence we are unaware of. A discussion on how to include an extra environmental variable is in section 6.4. All of data products that are used in this research project is in UTC (Coordinated Universal Time) time standard.

3.1.1 Land Surface Temperature

Bands 3-7, 13, 16-19, 20, 22, 23, 29, 31, 32 of MODIS are used to retrieve land surface emissivity, temperature and detect clouds. All of these parameters are needed for extraction of the LST products of MODIS [53]. There are some products that provide the user with the final parameter. For this study, the LST daily product has been selected; it is called *MOD11A1*. Here, we provide a brief description of this product.

MODIS is particularly important because of its global coverage, radiometric resolution and dynamic ranges, and accurate calibration in multiple thermal infrared bands designed for retrieval of LST, SST and atmospheric properties. Wan and Dozier [53] proposed a *Generalized Split-Window (GSW)* method for production of LST data that consists of the following main steps:

- Cloud masking: Cloudy pixels that are detected and kept out of LST production
- Estimation of atmosphere column water vapor and lower boundary temperature: It is estimated from seasonal and regional climatological data.
- Land surface types and fractional vegetation cover: The VNIR channels of MODIS and AVHRR are used to estimate land surface types and to derive NDVI. The fractional vegetation cover coefficient C can be estimated from the NDVI values

Finally, band emissivities can be estimated from fractional vegetation cover values pixel by pixel. Once emissivities are known, LST can be computed.

A level 3 (L3) product is a geophysical product that, unlike a level 2 (L2) product, has been temporally and spatially manipulated, and is usually in a gridded map projection format referred to as tiles. *MOD11A1*, is a daily LST product at 1km spatial resolution. It is obtained by mapping the pixels from the *MOD11_L2* products for one day on the sinusoidal or integerized sinusoidal projection [52]. The first product, MOD11_L2, is an LST product at 1km spatial resolution for a swath. This product is the result of the GSW LST algorithm. MODIS LST data are in Hierarchical Data Format - Earth Observing System (HDF-EOS) format. A typical filename of an MOD11A1 file is: **MOD11A1.A2002027.h20v09.005.2007117021342.hdf**.

MODIS filenames follow a specific convention to provide the user with the important information about the file. For example, the above-mentioned filename indicates:

- **MOD11A1**: product short name
- **.A2002027**: Julian date of acquisition (A-YYYYDDD)
- **.h20v09**: tile identifier (horizontalHHverticalVV)
- **.005**: collection version
- **.hdf**: data format (HDF-EOS)
- **.2007117021342:** production date and time (YYYYDDDHHMMSS)

Tile identifier is determined based on MODIS tiling system that is explained in 3.1.1. Every collection version share common characteristics such as spatial or temporal resolution.

MODIS tiling system

A sinusoidal grid tiling system is used for this product. These are $10^\circ \times 10^\circ$ tiles at the equator [16]. The tilling system for the entire planet is depicted in Figure 3.1.

The product MOD11A1 product of MODIS has a number of Scientific Data Sets (SDSs), that each one is a subdataset in the *.hdf file [52]. We are using *LST_Day_1km* and *LST_Night_1km* that are LST day and night data. The temperature per pixel of this product is the temperature at the time of acquisition. Generally speaking, Terra daytime pass is at around 10:30 AM and the night time pass is around 10:30 PM local equatorial time.

The temporal resolution of these data is daily but eight-day and one-month aggregated data are also available as alternative products. The summary of SDS information of the LST_Day_1km is as follows:

```
SDS Name: LST_Day_1km
Long Name: Daily daytime 1km grid Land-surface Temperature
Unit: Kelvin
Valid Range: 7500 - 65535
Scale factor: 0.02
Add offset: 0.0
```

So, the values should be multiplied by 0.02 to get the LST data in Kelvin, but there no need to add any offset. According to the study area geometry and location, the relevant tiles can be identified. This can be achieved by comparing the coordinates of study area and bounding boxes of the tiles. For instance, our study area fits well in h20v09 and h21v09 tiles. The corner coordinates of the corresponding tiles for all of the days and products are the same. The next step is to collect all the corresponding tiles throughout the entire period for the study area to make the analysis

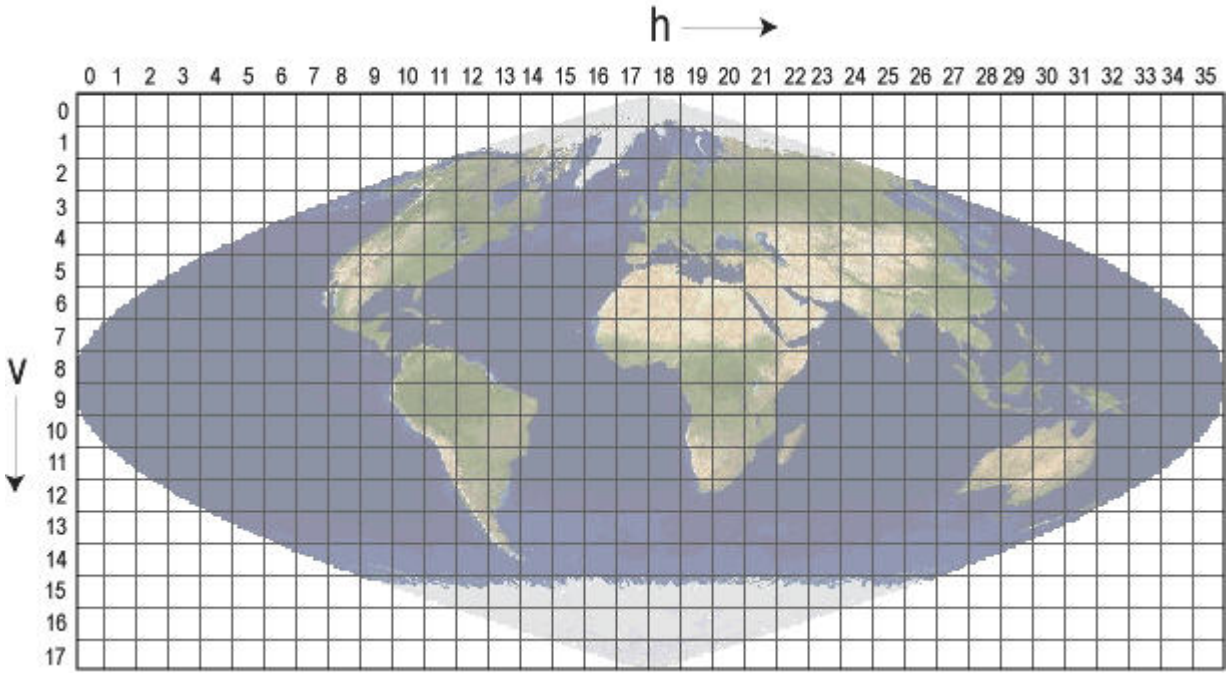

Figure 3.1: MODIS tiling system [16]

possible. I used the Python program in Appendix A to download desired data from USGS ftp site (ftp://e4ftl01.cr.usgs.gov). At the final stages of writing this thesis, we were informed that *NASA Land Processes Distributed Active Archive Center User Services (LP DAAC)* which provides these data is undergoing a transition from ftp to http (e4ftl01.cr.usgs.gov). Accordingly, after January 16, 2013, only http option was available. They asked LP DAAC users to adapt any automated script for data retrieval before the deadline. The updated function for http is also available in Appendix B. More explanation of the scripts is provided in Chapter 4 of the thesis.

3.1.2 Land Surface Air Temperature

Daily minimum and maximum values of *Land Surface Air Temperature (LSAT)* are widely used as an input in environmental applications like forestry and agriculture [17]. LSAT differs from LST in that it is the air temperature and cannot be recorded by satellite sensors. Satellite observations only account for measuring the surface temperature, not the air temperature. To acquire timely and continuous air temperature data throughout a region, interpolation between meteorological stations can be applied. Interpolation will result in reasonable information, when the stations or sensors throughout the study area are dense enough, this is not the case for most parts of the world, including our study area. The stations are typically separated by a distance of 30–50 km or more.

There is a number of studies that try to estimate daily minimum, maximum and mean air temperatures from MODIS LST data [43, 30, 41, 10]. Also, these statistical methods demonstrated high correlation between air temperature and surface temperature [30, 41], computing the air temperature from LST is beyond the scope of this study. This is mostly because the statistical estimates depend on many factors like altitude [30] and local vegetation fractions [43], and also by solving regression value for saturated NDVI [43], for example. Hence, extracting a robust algorithm for computing air temperature, to be put in the automated system is far from practical.

Nevertheless, according to [51] MODIS night products provide a good estimation of minimum air temperature, while the difference between LST and air temperature is more dependent upon the ecosystem, solar zenith angle and cloud cover.

3.1.3 Precipitation

The *EUMETSAT Multi-Sensor Precipitation Estimate (MPE)* for the last 24 hours is available at (oiswww.eumetsat.org/SDDI/html/grib.html). There are four files per hour, each one about 2.2 MB in size. To overcome the shortcomings of a single data source, EUMETSAT uses a combination of measurements from different satellite instruments for estimation of precipitation [23]. It results in high temporal and relatively high spatial resolution. The idea of developing MPE was to combine accurate instantaneous rain rate retrieval of the Special Sensor Microwave Imager (SSM/I) data, and high spatial and temporal resolution of METEOSAT IR-imagery. The MPE algorithm is integrated in the Meteorological Extraction Facility (MPEF), which is EUMETSAT's operational environment for METEOSAT data. The MPE data can be particularly useful for Africa to fill the gaps in their sparse ground base measurements [23].

The filename looks like **MPE_20121207_2215_M9_00.grb** which *20121207* is the date of data acquisition in *YYYYMMDD* format and *2215* is the time of data acquisition in hhmm format. *M9* shows that it is from *Meteosat 9* satellite which is one of *METEOSAT Second Generation (MSG)* satellites. Data are in the second edition of *GRIB* format that is called *GRIB–2*. *GRIB* is the name of data representation form for *General Regularly-distributed Information in Binary* form. The MPE is comprised of near real-time measurements in mm/hr rate for every METEOSAT image in its original resolution [36]. At ITC, one-day aggregated data are obtained from 15 minute products. The files are compressed and can be found in ftp website (ftp://ftp.itc.nl/pub/mpe/) of ITC and the aggregated data of previous day at 08:00 UTC is processed and provided half an hour later [36]. In this project, we use the daily aggregated data. They are in *ILWIS* format and a typical file name is: **fsummsgmpe20100103.mpr**. In this example, *20100103* means that the data is for January 3, 2010 (YYYMMDD). The data in this repository is available from 2010 onwards. The metadata of this file is:

```
Files: fsummsgmpe20100103.mpr
Size is 3712, 3712
Coordinate System is:
PROJCS["unnamed",
    GEOGCS["Unknown datum based upon the custom ellipsoid",
        DATUM["Not specified (based on custom ellipsoid)",
            SPHEROID["Custom ellipsoid",6378140,298.252981]],
        PRIMEM["Greenwich",0],
        UNIT["degree",0.0174532925199433]],
    PROJECTION["Geostationary_Satellite"],
    PARAMETER["central_meridian",0],
    PARAMETER["satellite_height",35785831],
    PARAMETER["false_easting",0],
    PARAMETER["false_northing",0],
    UNIT["Meter",1]]
Origin = (-5568748.275999999600000,5568748.275999999600000)
Pixel Size = (3000.403165948275700,-3000.403165948275700)
Corner Coordinates:
Upper Left  (-5568748.276, 5568748.276)
```

```
Lower Left  (-5568748.276,-5568748.276)
Upper Right ( 5568748.276, 5568748.276)
Lower Right ( 5568748.276,-5568748.276)
Center      (   0.0000000,   0.0000000) (  0d 0' 0.01"E,  0d 0' 0.01"N)
Band 1 Block=3712x1 Type=Float32, ColorInterp=Undefined
  NoData Value=-9.9999996802856925e+037
```

As is obvious from the metadata, the spatial resolution of the data is approximately 3 km, which is the same as the original MPE data.

3.1.4 NDVI

The *Normalized Difference Vegetation Index (NDVI)* is a well-known index that provides an image of relative biomass. There are a number of characteristics that makes NDVI useful for displaying greenness. Plant materials are highly reflective in the *Near-InfraRed (NIR*) band, and on the other hand, chlorophyll absorption in the red band is another factor that can be useful. This index takes the advantage of these two characteristics of near InfraRed and Red bands in a multispectral raster dataset [34].

NDVI is generally used for monitoring and prediction of agricultural production, drought, and fire zones. The NDVI is eventually a single-band dataset that shows greenery. The negative values represent water, cloud, and snow, and values near zero represent rock and bare soil. Moderate values correspond to grassland and shrub (0.2 to 0.3), while high values (0.6 to 0.8) indicates tropical rainforests (earthobservatory.nasa.gov/Library/MeasuringVegetation). This index outputs values between -1.0 to 1.0. NDVI is generally computed using the following equation [34]:

$$[\,\mathrm{NDVI} = \tfrac{(IR-R)}{(IR+R)}\,]$$

In this research project, the NDVI data that is provided by eMODIS, is selected as one data source. The USGS-EOS MODIS (eMODIS) system was designed, developed and implemented to meet the needs of a number of operational applications. eMODIS products provide a time series of weekly or 10 daily MODIS vegetation index and surface reflectance data. Its spatial resolution is 250m, 500m, and 1000m, but for Africa only 250m data is available. Its file format is geotiff. eMODIS has historical and expedited data in which historical data has been archived from 2000 onwards, and issue updates in less than a month, but expedited data are available from 2011 up to present and its updates are daily. This is for real time monitoring. It has a specific filename convention that make every file unique. For example, a typical filename for NDVI composite images is: **AF_eMTH_NDVI.2012.341-350.QKM.VI_NDVI.005.2012356022350.tif**. *AF* indicates that the data is for Africa but the rest follows the same routine as for other eMODIS NDVI Composite Images. The general template for eMODIS filenames is shown below:

```
RR_eMnn_parm.YYYY.DDD-DDD.SSS.BB_BBBB.VVV.yyyydddhhmmss.ext
RR     = Region, such as
         AK for Alaska
         AF for Africa
         CA for Central Asia
 US for continental United States

eMnn = short name, such as
eM for eMODIS
A for Aqua
```

```
T for Terra
H for Historical
E for Expedited

_parm = parameter, such as
REFL for Reflectance
NDVI for NDVI

YYYY = acquisition year

DDD-DDD = composite start-stop DOY range in Julian days

SSS = spatial resolution, QKM (250 m), HKM (500 m), or 1KM (1000 m)

BB = band number, such as
B1 for Band 1 surface reflectance (Red)
B2 for Band 2 surface reflectance (NIR)
B# for Band # surface reflectance
VI for vegetation indices

_BBBB = file description, such as
COMPRES for zip files
QUAL for quality
NDVI for NDVI
REFL for reflectance
ACQI for acquisitions image
ACQT for acquisitions table
META for metadata

VVV = version of MODIS surface reflectance code inputs

yyyydddhhmmss = production date and time (UTC)

ext = extension (.tif, .zip, .txt, .met, .jpg)
```

In Table 3.1, typical filename and description of one dataset are listed. *NDVI zip* is the file that is available as data source and its average file size is 1.6 GB. All the rest are the files that are extracted from this compressed file. All together the size of uncompressed files is about 4.5 GB.

We are using NDVI composite images in this project. The valid range of pixel values is from -1999 to 10000. Any NDVI computation less -1999 is assigned a pixel value of -1999. The pixel values should be multiplied by 0.0001, so that the values between -1.0 and 1.0 which are meaningful for scientific community are derived. The accessory files (acquisition image, acquisition table, quality, browse and metadata) are extracted from NDVI composite and copied into NDVI zip packages.

3.2 TOOLS

3.2.1 Python

Python is an open-source, modern, high-level programming language. High-level means that programs can be processed before they can run and this extra processing takes some time [15]. This cannot be considered as a disadvantage, because much time is gained in the programming itself, due to simplicity of programming in comparison to low-level languages. Besides, high-level languages are portable [15]. This means that with only a few modifications, they can run on different computers and platforms. This is not the case for low-level programs. Python has support for procedural programming, functional and object-oriented programming, and therefore it is used for a wide range of applications such as desktop applications, games, web-based systems and scientific programming. Simplicity of Python, in comparison to that of other high-level programming languages, gained popularity specifically amongst scientific programmers. Even though Python is an interpreted language and is criticized for being slow compared to compiled programming languages, Python's performance is amazingly good [54].

With Python standard libraries, performing tasks such as manipulating strings, mathematic calculations, compressing and decompressing files, downloading data from websites, and so on are easier. Python is not limited to its built-in modules and numerous numbers of custom modules are available for download and install, each for some specific application domain. Vector and raster geospatial libraries have strong support for Python. Some of these libraries are explained later in this chapter, and have been used throughout this research project. Simplicity of programming with Python, strong bindings with other geospatial open source libraries, and products and my own familiarity with it were the main reasons to choose this programming language.

3.2.2 PostgreSQL

PostgreSQL is an *Object-Relational Database Management System (ORDBMS)* and has a long history of development, which dates back almost to the dawn of relational databases. PostgreSQL is one of the most advanced DBMSs and the most advanced open-source DBMS in existence. In the standard PostgreSQL distribution the following features are found [14]:

- **Open-source:** the team of developers is an international team that contributes to the DBMS, and core members that are in charge of enhancement and optimization of it.
- **Standards compliant:** PostgreSQL follows the rules of SQL92 and SQL99 in most cases, except where unique features are not expressed by the standards.
- **Object-relational:** in PostgreSQL every table is a class and inheritance is defined between classes.

Table 3.1: Sample filenames and descriptions

File Description	Sample Filename
NDVI zip	AF_eMTH_NDVI.2012.341-350.QKM.COMPRES.005.2012356053712.zip
NDVI Composite Image	AF_eMTH_NDVI.2012.341-350.QKM.VI_NDVI.005.2012356022350.tif
NDVI Quality Image	AF_eMTH_NDVI.2012.341-350.QKM.VI_QUAL.005.2012356022350.tif
NDVI Acquisitions Image	AF_eMTH_NDVI.2012.341-350.QKM.VI_ACQI.005.2012356022350.tif
NDVI Acquisitions Table	AF_eMTH_NDVI.2012.341-350.QKM.VI_ACQT.005.2012356022350.txt
NDVI Metadata	AF_eMTH_NDVI.2012.341-350.QKM.VI_META.005.2012356053712.met
NDVI Browse	AF_eMTH_NDVI.2012.341-350.QKM.VI_NDVI.005.2012356022350.jpg

Other features of the DBMS are transaction processing, referential integrity, unique data types (such as arrays) and extensibility. Moreover, PostgreSQL has some unique features that are hardly found in other databases [42]. Some of these features are:

- **Multiple procedural languages:** none of the commercial and open-source databases can compete with the variety of languages PostgreSQL provides to write functions and aggregate functions. Built-in languages are SQL, PL/PGSQL and C. Additional environment installs, pave the way to make use of the commonly used PL/Java, PL/Perl, PL/Python, PL/R, PL/TCL and PL/SH.
- **Multi-column aggregates:** PostgreSQL has the ability to define aggregate functions that take more than one column.

3.2.3 PostGIS, spatial flavor of PostgreSQL

PostGIS is the spatial extension of the PostgreSQL that supports spatial object types like geometry, geography and raster. More than 300 spatial functions, operators, data types and indexing enhancements are provided by PostGIS [42].

PostGIS is built on other projects such as *Geometry Engine Open Source (GEOS)* to include advanced spatial operation support, *Open source Geospatial Foundation (OSGeo)* project and *projection support (Proj4)* [42]. PostGIS has some advanced capabilities that are listed here [42]:

- PostGIS has the ability to edit geometries by adding, removing and changing points, and by scaling, shifting and rotating the entire geometries.
- In addition to *WKT* and *WKB*, it has the ability to read and write geometries in *GeoJSON, SVG, KML* and *GML* formats.
- It makes use of spatial indexes and a complete range of bounding-box comparison operators to speed up queries by quickly identifying of matching features.
- It also makes use of libraries like *GDAL* and *PROJ* for spatial data manipulation.

A wide range of spatial functions and spatial comparison between geometries are provided in PostGIS [45]. Spatial functions in PostGIS can compute area, perimeter, length, distance, closest point, centroid, shortest connecting line, and more. Spatial comparisons include intersection, crossing, containment, equality, overlap, touching and so on. The functions and spatial comparisons consider the geometry's spatial reference, if known. In the case of spatial comparison, the spatial reference of two geometries should be the same. Despite the fact that PostGIS is not the only option to store geo-spatial data in a database, it is known as a geospatial powerhouse [54]. PostGIS has more functions and output formats than commercial offerings and its speed is evenly matched and is sometimes better than commercial ones for regular spatial needs.

3.2.4 GDAL

GDAL stands for *Geospatial Data Abstraction Library*, and is a translator library for reading and witting raster geospatial formats (www.gdal.org). It is released under an *X/MIT style Open Source license* by the *Open Source Geospatial Foundation (OSGeo).* It supports a wide range of raster formats that can be found in (www.gdal.org/formats_list.html). By default, GDAL supports almost 130 raster file formats. Not all of them can be written using GDAL. All the information about specific formats and their capability for reading and writing are in the list of formats.

A separate *OGR* library is intended to deal with vector formats. However, now both are partially merged and are called GDAL. Some sources refer to it as *GDAL/OGR* to avoid confusion. OGR, on the other hand, supports more than 70 vector file formats.

There are a number of drivers that make the reading and sometimes writing of various types of raster data possible. The appropriate driver is selected automatically by GDAL, when it tries to read the raster. The driver should be selected by the user at the time of writing.

GDAL Design

Figure 3.2 illustrates the GDAL model for describing raster geospatial data.

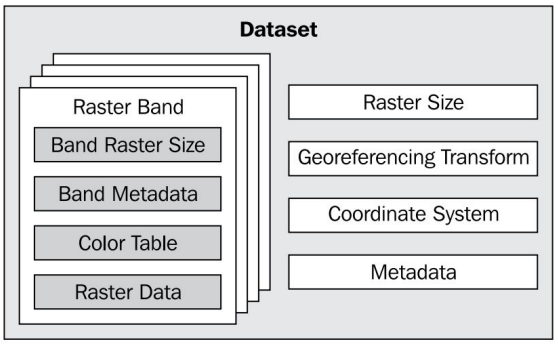

Figure 3.2: GDAL model [54]

Different parts of the model are described here [54]:

- **Dataset:** represents all raster bands and all the other information that is associated with them.
- **Raster band:** each raster is comprised of one or more raster bands, layers, or channels within the image.
- **Raster size:** is the number of rows and columns in the image.
- **Georeferencing transform:** makes the connection between georeferenced coordinates and raster coordinates. Two types of georeferencing transform are supported by GDAL, namely, ground control points and affine transformation.
- **Coordinate system:** describes the georeferenced coordinates that are created by the georeferencing transform. It has information about the projection and datum, and also the units and scale used by the raster data.
- **Metadata:** is the additional information for the data, so that it is more usable. Information like pixel size, corner coordinates, the time of acquisition are parts of metadata.

  Each raster band has the following components:

- **Band raster size:** is the number of rows and columns in each raster band. This may or may not be the same as the raster size for the dataset.
- **Band metadata:** some bands need extra information specific to that band.
- **Color table:** the way the pixel values are translated into colors.
- **Raster data:** the raster data itself.

Chapter 4
Design and implementation of a climatic database

During this research project, we want to build a system that digests the data from different sources, processes these and puts them into a spatio-temporal database. To do so, a number of steps should be followed from acquisition and digestion the data to loading these into the database automatically. As we want the system to be fully automatic, all the steps were programmed and stitched together to build one functional system. The scripts were written in Python and connection to the database was made through a Database Abstraction Layer (DAL). The scripts leveraged Geospatial Data Abstraction Library (GDAL) and a number of other libraries, to help the implementation of the scientific concepts.

Each piece of code was written as a function, so that it is callable and extendable. Functions that are related to one another formed classes, so that they can be imported from other scripts. The designed system consists of a number of main classes and some scripts that call those functions in the classes to execute the required operation. All the procedures to build such a system are described in this chapter. The database is meant to be updated as the data in the data source is being updated. We designed the application in a way that whenever the database is empty it builds the required tables, acquires the whole data available in the data source, performs the necessary processes, populates the database and makes it ready for further analysis. Whenever the database is not empty, but it has not been updated, it checks for new data in the data source and performs all processes for that new data. Needless to mention that in case the data in the database and data source are equally current, the application leaves the database unchanged. The application will be prompted periodically, using a command line level scheduling program.

4.1 DATA ACQUISITION

To acquire timely information, there should be an application that downloads the data automatically. Our data sources are stored and issue updates via ftp or http sites. Download techniques differ slightly for each protocol. Therefore, two classes were defined with necessary functions to handle them. Besides, the architecture of the data source affects the algorithm of the program slightly. Accordingly, each data source needs special treatment when it comes to automatic acquisition of the data. Nevertheless, no important changes are needed when a new data source is included in the system's data source list. A discussion on how to include an extra environmental variable or acquire data from other sources is in section 6.4.

4.1.1 Download from ftp server

To facilitate automatic downloading of data from ftp servers, the *downloadFTP()* class is defined, and which contains a number of functions. The script is available in Appendix A. There are two main functions in this class. All the other functions are used in the two main functions. The function *mod11a1_All()* is defined specially for downloading *MOD11A* series of products from USGS ftp server. As discussed in the previous chapter, the products are tile-based and the user

should define the specific tile identifiers that cover the study area. So, tile identifier should be a parameter. Start and end date are other important parameters that the user must define.

In the case of this system, start and end dates are defined automatically by the application following the database contents and data source updates. In the following sections, more elaborate explanation of the date management issue are discussed. The function compares the lists of the dates to download (according to the data source and required data), skips the missing dates and downloads the data for available dates to a directory that user defines. The downloaded data are in HDF format and has a number of subdatasets. In the next sections, we discuss how data should be extracted. As mentioned previously, we came to know that LP DAAC was undergoing a transition from ftp to http in early 2013, and ftp option would stop functioning after mid-January 2013. This means that our *mod11a1_All()* function would not be useful any longer. Later in this chapter, the adapted function for the http option is explained.

The function *perc_All()* does the same job for precipitation data from the ITC ftp server, except the data does not have tiles, and each file represents the data for the entire planet. The data are compressed files in ILWIS format.

4.1.2 Download from http server

Some data sources are only available via the http protocol. Another class was defined to handle automated downloads from http sites. The class is called *download_url()* and includes two main functions. The entire class scripts are available in Appendix B. One to download NDVI data from (dds.cr.usgs.gov/emodis) is called *NDVI_All()*. The other to download LST data from (e4ftl01.cr.usgs.gov) and it is called *LST_All()*. The latter is needed because of the mentioned LP DAAC transition from ftp to http.

The function *NDVI_All()* inputs the directory to download, and start and end dates. The start and end dates are the intervals in which the data is downloaded. Using the information from the user side for the start and end dates, the script tries to open the relevant folders in the website, read them and write the files into a folder in the computer or network.

Since the files to download may be big in size (in this case 1.6 GB), python throws a memory error sometimes. To deal with this issue, I cut the files into chunks when downloading. It starts downloading every chunk at a time. This is the technique that I imagine professional download manager software packages use to make the resumption capability possible.

The function *LST_All()* does the same job for the LST data of MODIS. Whenever there is no file available for a specific date, it does not throw an error and just informs the user and continues downloading the rest.

All the download programs need to be adapted with the format of the piece of code that determines the last file in the data source and last file in the database. So, I designed all of the programs with one standard output format. They have *start year* and *start day of year (start_doy)* as the start date, and *end year* and *end day of year (end_doy)* as the end date.

4.2 PREPROCESSING

The raw data should undergo some preprocessing, so that it can be put into the database. For example, we have to be sure about format compatibility and spatial reference system. The data also has to cover the study area and be cropped according to the extent of study area. All functions that help to convert the raw data ready for import into the database, are in a class called *Tran()*. The scripts of the class can be found in Appendix C.

4.2.1 Unpacking

Some of the data in the data sources are compressed files. To make use of the data, these files should be uncompressed, and put in a separate folder. To do so, Python has built-in modules to deal with compressed files. A function has been defined for unzipping the compressed files. It primarily lists all the files in a specific folder, and then tries to uncompress the files. This makes use of the *zipfile* module in Python. Whenever the file is not a zip file, it just skips it. The function is called *unpack()* and is part of the *Tran()* class. This is the case for NDVI and precipitation data in this study. Uncompressing NDVI files results in six files of which only one is used in subsequent processes. None of the files are dependent on another, so all other files can be deleted. A detailed explanation of the files is provided in Chapter 3.

Precipitation files are also compressed files, but the containing files are in the ILWIS format. They are five files and all of them are needed for processing. Only **.mpr* files are introduced in the scripts, but they make use of other accompanying files. There are two files that are responsible for georeference information, namely, *mpe_georef.csy* and *mpe_grf*. They are the same for all the data files and are overwritten, when the files are uncompressed into one destination folder.

4.2.2 Format conversion

The NDVI files are in the *GeoTiff* format and there is no need for extra format conversion operation. It can be readily used for subsequent processes. The ILWIS format is also one of the raster file formats that has full read/write support according to the GDAL raster formats list (www.gdal.org/formats_list.html).

The HDF format has a number of subdatasets that has to be converted to one of the supported formats in GDAL. It can be done by a command line level GDAL operation. More specifically, it is done by calling *gdal_translate* into the Python script. By default, it translates every format to GeoTiff format. The function *conv_format()* is defined inside the *Tran()* class to extract *LST day* and *LST night* subdatasets and transform MODIS LST data. The translated data would have the same name as the **.hdf* files, except that at the end of the filename a *_day* or *_night* suffix is added. The function lists all the files in a specific folder and then translates them into GeoTiff format.

4.2.3 Crop the study area

It is important not to import all the available data, but only that part of the data that is related to the study area. Accordingly, it would be much better to delete all the information outside the study area. Cropping accomplishes this by comparing the borders of the study area with the available data. It is important that both data, one in raster format and the other as a vector file, have their spatial reference system in common. Otherwise, GDAL cannot compare them. Either the raster or vector file (or both) should be transformed to a common spatial reference system. This choice is discussed in detail in upcoming chapters.

GDAL 1.8 introduces the *crop_to_cutline* option in its *gdalwarp* command, so that it crops a raster file by the extent of the *cutline data source*. It is an *OGR* supported format of which the extent is the border of the study area (www.gdal.org/gdalwarp.html). The cropped file has the same filename as the original file, except for a *_cropped* suffix.

I added the *-dstnodata None* option, so that whenever no data is recorded for a specific pixel, in the output cropped file null values will be inserted for that pixel. The pixels with no data sometimes constitute the majority of a raster file, so they deserve attention, both in data manipulation and subsequent calculation.

The functions *crop_area()* and *crop_area_NDVI()* in the *Tran()* class are responsible for the above-mentioned tasks. They list all the relevant files in a folder and start processing using one

command line level *gdalwarp* operation for each file.

4.2.4 Automated preprocessing

Above, all the necessary functions on the raw data are defined. The problem is that not all the operations are needed for each product. For example, LST data are not compressed, so there is no need to uncompress. We defined other functions for each class of dada, to call the necessary functions in sequence. The function *raw_hdf_2db()*, inputs the study area shapefie name, and the choice if the user wants to operate on LST day or night products. It primarily extracts the required subdataset (day or night) and converts it to geotiff format, using *conv_format_day()* or *conv_format_night()* functions. Subsequently, it crops all the converted data sets according to the study area shapefiles. It imports the files as raster data into the database using *R2dbLST()* and *import2DB()* functions. This function will do all the necessary processes to import the downloaded raw MODIS LST data into a relevant place in a database.

The function *raw_perc_2DB* does the processes for precipitation data. Precipitation data needs to be unpacked because it is in the form of compressed files. It results in ILWIS files as explained earlier. The ILWIS format is one of the GDAL supported formats, and can be cropped using a shapefile of the study area. It should first be noticed that the shapefile should be transformed to the spatial reference system of the raster files. The functions *R2db()* and *import2DB()* are subsequently used to digest and import the cropped precipitation files into a database.

Function *raw_NDVI_2DB()* does the same operation and procedure as the precipitation files. The only difference is that as it is mentioned earlier, the average size of NDVI files is 1.5 GB. The size of uncompressed files is three times as much. Together they make about 6 GB of data. Not all of the files are needed and only one of the uncompressed files participates in the operation. To make the system more memory efficient, during the processes, it is better to remove all the unnecessary files immediately after unpacking. This is particularly important at the time of building the system for the first time. This prevents the system from overloading a lot of unnecessary data.

4.3 IMPORT TO THE DATABASE

After cropping, the data is available for import into a database. The raster format is chosen as data structure throughout this thesis, but a discussion on the choice of data structure in the thesis is available in Chapter 6. The database schema for this project is in Figure 4.1. The classes *LST_day, LST_night, NDVI,* and *Precipitation* forms the main classes in the schema. They inherit their attributes and operations from a super-class called *CLIMATE_PARAMETER*. The class *envvar2table* that makes the connection between the table names and the climatic parameter, such that instead of inputting a table name the end-user inputs a valid environment variable. A more elaborate database schema that supports a data aging mechanism is described in the next chapter.

The extension *PostGIS Raster* helps in developing the application. PostGIS Raster uses the GDAL library to manipulate raster data. To this end, it supports any kind of raster file format that the GDAL library supports. One particular GDAL install may not support all the raster formats. To verify the supported raster file formats by the particular GDAL install, the following command can be used:

```
raster2pgsql -G
```

The result of the command can be found in Appendix F. Through the use of the *raster2pgsql* command, the raster files can be imported into a database. The example of general format of using this raster loader is as follows:

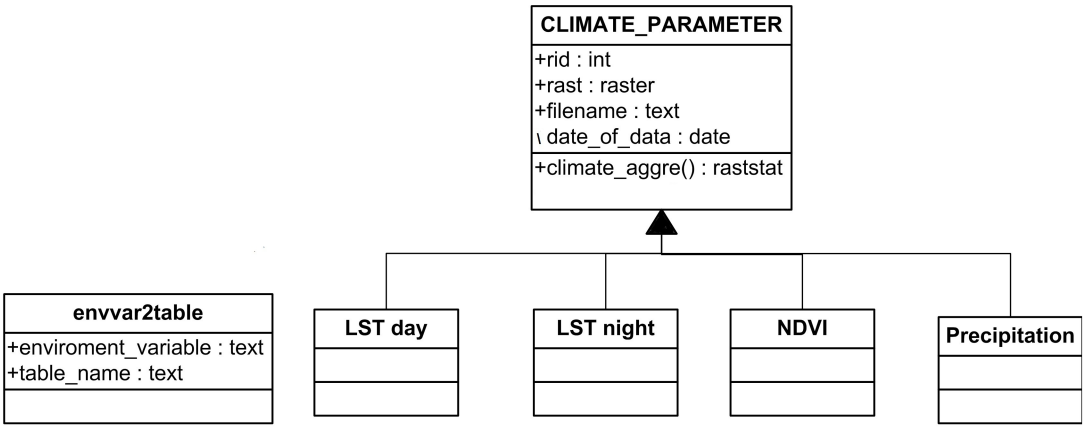

Figure 4.1: The database schema

```
raster2pgsql raster_options raster_file myschema.mytable > output.sql
```

To load raster data, it is important that we make use of *raster2pgsql* with proper options. The command *raster2pgsql -?* will display a help screen that includes all the options. The options that are used for importing data and also the ones that can be useful later in this project are: *-s,-t, -R, -a,* and *-F*. The full explanation can be found in different sources and also the help screen. Here just some of the issues regarding their usage that affected this research project are mentioned.

The option *-t* determines the tile size. It cuts the raster into tiles to be inserted one per table row. Since the study area is small and the cropped raster files are not big files, in most cases we do not use this option. If the original raster file size was big and the user did not specify a tile size, PostGIS Raster divides it into tiles and every tile will be in a separate row in the table. Tile size is in $width \times hight$ format and is usually something like, 50×50, 100×100 or 200×200. The idea of tiling comes from the fact that smaller raster files are easier to deal with, when only small part of the raster is needed at a time [42]. No more than 100×100 tiling is recommended if lots of processing and analysis are expected to happen in the database.

The option *-R* registers the raster as an out-of-database raster. There are two options for storing data in a raster database using PostGIS Raster, namely, *in-database storage* and *out-of-database storage*. In the in-database method (that we are using), the full raster file is stored a record in a column or cut up the raster files into tiles and store each tile as a separate record. In the out-of-database option, only the geographic context associated with the raster files or tiles are stored in the database, and the corresponding pixel data in the file system. The absolute path to the file should be provided. We are not using this option, because of the following reasons [42]:

- It is not well tested at this point.
- The raster files can get deleted or moved from the file system.
- The analysis of raster files, such as reading pixel values and forming polygons is not fully supported.

The *-a* option, appends the raster file(s) to an existing table. The table and schema name should be exactly the same. This option is used for this project, since automatic updates requires appending the newly updated files to the existing table.

The *-F* option, add a column with the filename of the raster. We are using it because the filenames contain useful information that are needed in the database.

In the example of the general format of the command, it creates an ouput.sql file that would be saved in the same folder as the command is running. This file contains all the necessary commands to put the raster files into the appropriate tables. It is important to help these commands run in the server side DBMS, which here is PostgreSQL 9.2. It can be done by psql, the PostgreSQL interactive terminal.

4.4 DATE AND DATA MANAGEMENT

Since the system is dynamic and is updated automatically, the last updated data in the database and data source are important. Last updated record in the database can be easily determined by connecting to the database and running a simple query to identify the newest data. This requires that a table that provides the data acquisition dates that is available in the database. By default, such a column does not exist in any table of the database. Last updated files in the data source can be obtained by connecting to the data source location, finding the last file and extracting date of data that is usually embedded in the filenames. This is done before the data acquisition by connecting to the data source server. In this section, all the functions are explained. Every data source has different architecture to store data, and finding a generic solution for all the data products and data sources is far from practical. A class called *date_func()* is defined that contains all the necessary functions. It can be found in Appendix D.

4.4.1 Last data in the data source

For NDVI, *LastNDVI()* function is used to output the date of data acquisition for the newest data available in the data source. To do so, the url that contain all the filenames with its updates should be opened and read. Subsequently, the year and day of year can be extracted from the filename of the last file. The output of this function would be the year and the day of year of the newest file in the NDVI data source. The last date is different for the case of historical and expedited data. Expedited data are more updated and are used for real-time analysis, so they are used for updating the database. *LastLST()* and *LastPerc()* functions do the same job for LST and precipitation data. A new *LastLST_http()* function has been defined to adapt with the change in LST data sources. The change as it is discussed earlier is due to the transition of LP DAAC from ftp to http in final stages of implementation of this system.

4.4.2 post-processing

As is mentioned earlier, dates of data acquisition are imperative for further analysis. It would be much better if a separate column containing the dates of data is added in every main table of the database. This can be accomplished by paying attention to the filenames and the format in which dates of acquisition are stored. A simple SQL set of commands can create a column and populate it with the dates of data.

As the system is supposed to be fully automatic and the data is on the database side, any commands to manage the dates have to be run as SQL commands on the database. It would be much better if the SQL commands can be managed from within the application. Throughout this research project, for connections between the scripts in Python and the database in PostgreSQL, we make use of the Database Abstraction Layer (DAL). DAL is an API that maps Python objects into database objects such as tables, records and queries [13].

It can easily connect to most mainstream commercial and open-source DBMSs. For PostgreSQL, the following connection syntax is used:

```
db=DAL('postgres://username:password@host_name/database_name')
```

the variable *db* is a local, and stores the connection object DAL. There are a number of methods associated with DAL to manipulate data in the database. As an example, one of the methods is *define_table()* for defining a table with a number of columns. Another method that is used mostly in this project is *executesql()*. It allows to explicitly issue SQL statements from within Python. Accordingly, every SQL statement that is run successfully in the database itself, can be called using *executesql()* method. It also makes it possible to put every word in the SQL command as a parameter and build functions based on that.

The functions *fillDate_LST()*, *fillDate_Prec()* and *fillDate_NDVI()* are responsible for populating a column, which is called *date_of_data*. Here with dates of acquisition of data. It takes into account the filenames and the format in which the dates stored. It makes use of DAL to connect to the database from Python.

4.4.3 Last data in the data source

The system should eventually issue updates based on the newest data in the data source and the newest data in the database. The last updated data in the data source was discussed above. To find the last updated data in the database, it is enough to order the date column in the descending order and choose the first one. It is also called by *executesql()* method of DAL. Function *lastDB()* function will perform this operation. It inputs a table name as a string and outputs the last date as a tuple that contains year and day of year (doy). It throws an error when there is no such table. The error means that the system has not built, and it wants to start building from scratch, so the table does not exist. In this case, it should start from the earliest data. We know that there is no data before the year 2000 in any of the data sources. So, in the function the beginning of this year is introduced as the date the system should start building up.

4.4.4 Handling unexpected errors

After preliminary tests and establishment of the system on the server, the system should work automatically and issue updates on a regular basis. The automatic program that runs on a server machine may throw errors because of all sorts of unexpected reasons such as a change in source data architecture (for example, change in USGS http server for LST), a problem in Internet connection or connection with the server or some viruses in our own server that make changes to the files. The administrator of the system should be aware of these errors. For this system, an automatic e-mail service has been designed to send error messages with its trackbacks automatically to an e-mail account immediately and asks for their attention.

The method *smtplib()* makes it possible to connect to an e-mail account and send messages through it. It requires *SMTP (Simple Mail Transfer Protocol) outgoing server and port* for a specific e-mail provider. For example, for a Google mail account, *smtp.gmail.com* can be use as the SMTP server and *587* for the SMTP server. As long as the program will be run on ITC server and automatic e-mails cannot be sent from this network, we used a command line Linux functionality to send e-mails with error messages, from a Linux account to an e-mail address. A function called *send_error()* in *date_func()* class is defined to send error messages from a Gmail account. The function is not used for the case of this project and is for the case that the program does not run from a Linux account in a network.

4.5 AUTOMATED APPLICATION

The functions that are explained make it easier to perform the automated processes, but they cannot output the expected outcome in isolation. Another program that calls each function carefully with its specific parameters is required. All the procedures are explained here in sequence. The scripts for performing these processes for one of the parameters (LST) are in Appendix E. The Framework work flow and data flow is illustrated in Figure 4.2.

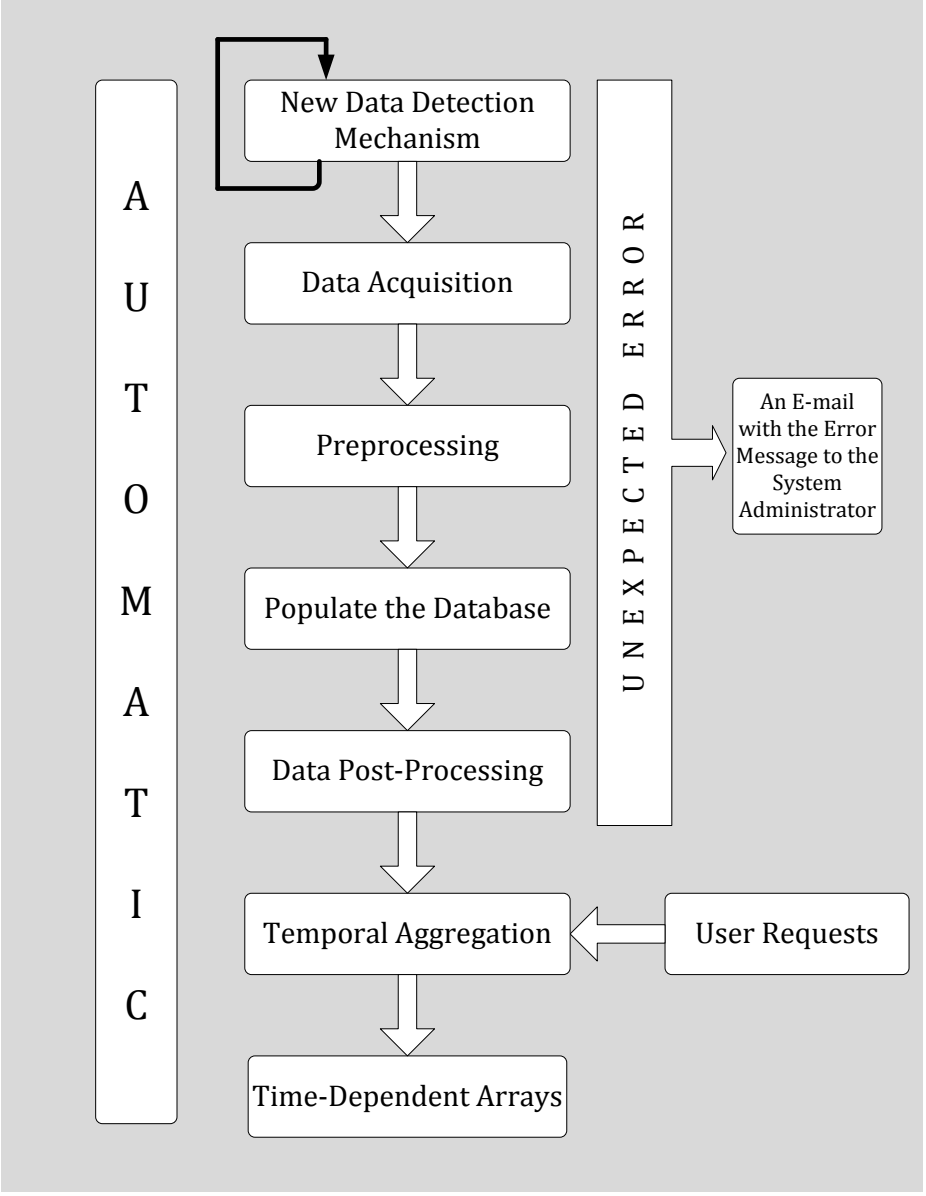

Figure 4.2: The system work flow and data flow

The script primarily creates a folder in the same folder as it runs if such a folder does not exist, to do all the processes on the files there. Subsequently, it tries to create the required tables and columns if it is the first time the system runs. All the main tables contain the following columns: *rid*, *rast*, *filename* and *date_of_data*. Attribute *rid* is a *serial integer* and it is the primary key, *rast* contains the raster files while its data type is *raster*, *filename* is the full filename of that raster and

its data type is text. Attribute *date_of_data* is supposed to have dates of data acquisition extracted from filenames and its data type is *date*. If such a table already exists, it does not make any change in the database.

Now it is the time for start data acquisition from relevant sources. The functions that facilitate the download of data from sources need the range in which data should be downloaded. The range can be determined by the newest data in the data source and the last data in the database. If there is new data in the data source that has not been absorbed into the database, the script downloads this. Otherwise, it does not make any changes.

After downloading and saving the files, the system begins to perform preprocessing on the data. This may include uncompressing the files, format conversion and cropping. The final files are imported into a specific table in the database. For every parameter, there is a predefined function in *Tran()* class to do all the necessary procedures. There are some input parameters that should be defined before running the program. For example, the name of the shapefile of the study area extent should be defined as well as the database connection information.

The idea is to update the table for the newly added data and prepare it for further analysis such as aggregation. By executing the *Tran()* class, *rid*, *rast* and *filename* columns will be updated, but not The *date_of_data* column. So, it should be updated using the functions that are defined to do this. After this stage, the tables should be filled for the missing dates, as explained earlier. This can be achieved by *fillMissing()* function. By missing dates, we mean the dates that there is no data available on the server, and accordingly, there is no record for that date in the tables.

The automated system may throw errors and stop execution due to unexpected errors. As explained above, the system administrator should be aware of these errors and the automatic e-mail system will notify the administrator of the error. Here, the system will try to download the data, preprocess and put them into a database and prepare the database for further analysis. If it fails to do so for any reason, it sends the error message to an e-mail account. At the end, all the downloaded files and also the files that are produced in the operation must be deleted from the computer. Our data sources issue updates every day, but we trigger the automated program twice a day using *crontab* functionality in Linux to make sure that the database always remains updated.

Chapter 5
Aggregation and data aging

Data that are gathered and stored using the system are not useful for the end user in isolation. According to the requirements of the user, some aggregated levels of the data should be extracted from the system. This is because the climatic information is meaningful only when it is interpreted as a combination of a number of records. For example, for historical analysis of the data, weekly temperature or rainfall may contain more important information than daily data.

Managing large amounts of data that are updated continuously is another issue to be taken into consideration. Data that is important at a particular time may not be that important after some time. It is in the interest of the database administrator, to gradually aggregate or remove the less important data from the database. The process is called *data aging* and a method is proposed below to deal with this issue.

5.1 AGGREGATION

Summarized information in a continuously updated databases such as the designed system for this research project, can be more important than the recorded data. Aggregation results can be a function of different parameters. They can be dependent upon the aggregation level, the time period, the climatic parameter or the aggregation function that is suitable for the parameter, and the location.

A function that can input all the mentioned parameters and output the aggregated data is defined. The function is responsible for most of the weather statistics that a user may need. The defined function is:

```
CREATE OR REPLACE FUNCTION publicdata.climate_aggre
                           (envvar text, x double precision,
                            y double precision,
                            src_srs integer, trg_srs integer,
                            start_date timestamp without time zone,
                            end_date timestamp without time zone,
                             agg_level integer)
  RETURNS SETOF record AS
$BODY$WITH
P as
( SELECT st_transform((st_GeomFromText
                     ('POINT(' || $2::text || ' ' || $3::text || ')',$4)), $5)
                      as point),

D as
( SELECT (publicdata.get_envvar_vals_for_pt($1, P.point)).doy,
         (publicdata.get_envvar_vals_for_pt($1, P.point)).val
```

```
  FROM P),

F as
( SELECT all_dates.date_col, min(D.val) as minval
  FROM ( SELECT date_trunc('day', dd):: date as date_col
         FROM generate_series($6::timestamp, $7::timestamp,
                              '1 day'::interval) dd) as all_dates
         LEFT JOIN D ON all_dates.date_col =D.doy
         GROUP BY date_col
         ORDER BY date_col )


SELECT date_col AS greatest_id_in_group,
       avg_last_window_inclusive,
       stv_last_window_inclusive,
       cnt_last_window_inclusive
  FROM ( SELECT date_col,
                avg(minval) OVER (ORDER BY date_col ROWS ($8-1) PRECEDING)
                            *0.02-273.15 AS avg_last_window_inclusive,
                stddev(minval) OVER (ORDER BY date_col ROWS ($8-1) PRECEDING)
                             AS stv_last_window_inclusive,
                count(minval) OVER (ORDER BY date_col ROWS ($8-1) PRECEDING)
                             AS cnt_last_window_inclusive,
                (((to_char(date_col, 'J')::integer)-
                (to_char($7::date, 'J')::integer)))%($8) as m
         FROM F ) as x
  WHERE m % $8 = 0
$BODY$
```

The function inputs a climatic parameter, the coordinates of the point, the source and target spatial reference systems, start and end date, and the aggregation level. For example, the user may want to extract 10 daily aggregated data from LST day data between 2010-01-07 and 2011-05-31 for a point with the following coordinates in WGS84 system:

```
Latitude: -1.32
Longitude: 30.42
```

The function call for the period is:

```
select publicdata.climate_aggre
('LST day', 30.42,-1.32, 4326, 35991, '2010-01-07', '2011-05-31', 10)
```

It returns the dates and aggregated values. Part of the answer is:

```
(2011-02-20,32.794,193.300,5)
(2011-03-02,28.39,176.128,4)
(2011-03-12,33.743,175.597,3)
(2011-03-22,,,0)
(2011-04-01,26.636,49.662,3)
(2011-04-11,23.720,246.780,2)
(2011-04-21,22.030,79.195,2)
```

The meaning of elements in the results where we use (the second line) is:

- **2011-03-02:** the aggregation happened in the period of ten days between '2011-02-21' and '2011-03-02', and for all the data available in this period the aggregated values are computed.
- **28.39:** the average of all the values in the period for the point is $28.39°C$. The average function is just an example and other aggregate functions can be used. The other aggregate functions that can be used are: max (maximum), min (minimum), and sum.
- **176.128:** is the standard deviation of the measurements in the period. The aggregation function that is used is *stddev* (sample standard deviation of the input values).
- **4:** The number of recorded data in the period is four. For all the rest, no data is recorded. The aggregation function that is used is *count* (number of not none values).

The function makes use of a number of *CTEs (Common Table Expressions)* to result in the desired outcome. Here the CTEs are explained:

- **P:** extracts the dates and values for all of the available data in the table.
- **F:** generates series of dates between the start and end dates, joins it with *D* and outputs a table with the dates and values between start and end dates. Generating date series is important when there are some missing dates in the tables. It lists all the dates and joins it with *D*. All the dates between start and end date with the values from *D* result. Missing dates certainly have null pixel values.

  The variable *minval* may be confusing and needs an explanation. In the case of LST data, more than one tile covers our study area. It may be the case for other data sources as well. Accordingly, there is more than one record for every date. A single point in the study area, would fall in just one of the tiles and the value for the others would be null. The aggregation function *min* is used just to extract the *not null* values. Some of other aggregation functions such as *max* and *avg* would do the same job.

Bear in mind that the missing dates are already filled at the time of building the database and the date series part is not needed, but it is added to the query to make sure that it is general and it results in reliable answer even if the table includes some missing dates.

The result of *F* is not the required answer. The user wants the values at N days aggregated level. The query should be partitioned into N-day groups. This can be done based on the *day of year (doy)*, but a problem arises in the turn of every year. The problem is that the N-day groups may not coincide perfectly with the start date or the end of the year, and also the grouping would start from the first day of the next year if the *doy* is the basis for calculation. This may not exactly be what the user wants from the aggregation. So Julian day of the dates are subtracted from the Julian day of the start day and this is used as the basis for partitioning. The query considers every row plus N-1 preceding rows as one group and performs the aggregation function over it.

Another function has been defined to compute the aggregated values of the same time of the year, over multiple years. The function is in Appendix H and is called *climate_aggre_Multiple_Years()*. For example, a typical function call is:

```
select publicdata.climate_aggre_Multiple_Years
('LST night', 29.4,-1.66, 4326, 35991, '2002-02-02', '2012-06-03', 5)
```

This function computes the five-day aggregate data for the early June and for the period of almost ten years. The result of this query is:

```
(155,2012-06-03,14178.0,10.14)
(155,2011-06-04,13996.75,141.21)
(155,2010-06-04,14105.0,)
(155,2009-06-04,13440.50,679.52)
(155,2008-06-03,14202.0,21.21)
(155,2007-06-04,14007.67,336.69)
(155,2006-06-04,14081.50,116.04)
(155,2005-06-04,14112.50,144.95)
(155,2004-06-03,14062.50,2.12)
(155,2003-06-04,,)
(155,2002-06-04,,)
```

The first element is the *day of year (doy)* which is the same for all years. This means that from 2002 to 2012, the aggregate value of days that their doy is between 151 and 155 is computed. The second element is the date for this doy. The third is the aggregate value and the last is the standard deviation.

5.2 MOVING AGGREGATE VALUES

It may be important for the user to know the moving aggregate values over time. For example, imagine for every day one needs to know the average temperature in the last 10 days. This is the moving average of the temperature over 10 days. A more generic function has been defined to address this problem. Instead of the aggregation level, the function inputs the number of preceding and following days. For instance, for computation of the last 10 days the number of preceding days is 9 and the number of following days is 0. The function looks like the previous function with slight changes. That's why we put it in Appendix H. A typical function call for this function is:

```
select publicdata.climate_aggre_mov
('LST night', 30.4,-1.6, 4326, 35991, '2000-06-02', '2000-06-21', 3, 3)
```

For the defined period in time and location, it builds a window of seven days for every day. In other words, it computes the average value taking into account the last three days, the current day, and the following three days.

```
(2000-06-05,14479,14413.5,74.0292734891632807,4)
(2000-06-06,,14386.2,88.5251376728666828,5)
(2000-06-07,14308,14396.0,96.0130199504213074,5)
(2000-06-08,,14384.0,106.448735705659,4)
(2000-06-09,14277,14369.75,92.3773962251228065,4)
(2000-06-10,14472,14369.75,92.3773962251228065,4)
(2000-06-11,,14397.75,84.0173593173855153,4)
(2000-06-12,14422,14371.60,93.3450587872759337,5)
(2000-06-13,,14403.40,79.0493516734957163,5)
(2000-06-14,14420,14409.60,86.6273628826365694,5)
(2000-06-15,14267,14409.60,86.6273628826365694,5)
(2000-06-16,14436,14406.50,99.7079067409734897,4)
```

The first element is the date, the second is the extracted value for that particular date, and the third is the aggregate value. This can help the end-user to draw diagrams based on the values

extracted from the database more smoothly. The fluctuation of the data may be so much that its readability is affected. Besides, there are many missing data in some data sets. Displaying data that contains missing data as diagrams may not be useful for the user. The user, looks for trends and patterns. They cannot be found when the diagram is not continuous.

Moving average with the window size of N is defined, for every record compute the average over (N-1)/2 preceding record(s), the record itself and (N-1)/2 following record(s). It fills most of the missing values with approximate values, taking into account the values from the neighboring records. It is important to notice that the window size cannot be an even number. The last example is a case of moving average for smoothing when the window size is seven.

To compare the results visually, for a course of almost six months, recorded, moving average data and aggregated data is illustrated in Figure 5.1. The blue lines show the recorded values and as is obvious from the diagram, following the blue line trend is difficult. But they consist of the original recorded data and are more precise certainly.

The red line provides the moving average data for the same period in time with the window size of seven. The user is able to follow the line to find a particular trend or pattern. How accurate and useful this method can be and the optimal window size is the subject of future research.

The green line shows the seven-day aggregated data as another means of approximation and smoothing of the diagram. Aggregated and moving average values seem to follow almost the same trend. Seven day aggregated data represents a single value for a period of a week, but moving data has a value for (almost) every day. The choice of the methods depends on the requirements of the user and has to be the subject of a complementary research.

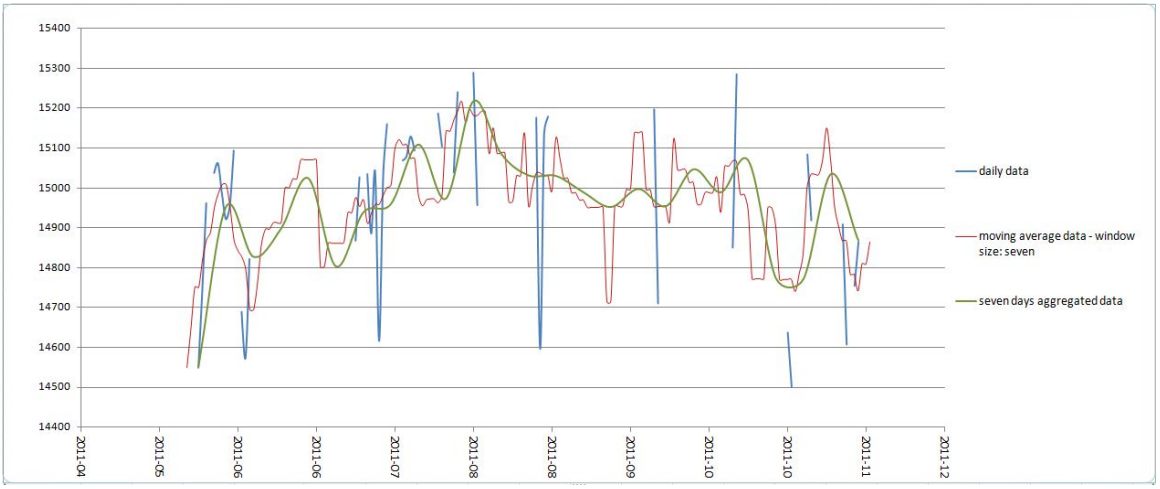

Figure 5.1: Comparison of trends for the measured, aggregated and moving average values

5.3 VALIDATION OF THE RESULTS

In addition to daily data, eight-day and monthly aggregated data is also available for LST data. Its spatial resolution and map projection is the same as the daily product and its *Earth Science Data Type (ESDT)* is *MOD11A2*.

When data for a specific date is published as an eight-day aggregated data, it includes the data from the beginning of that day plus the following seven days.

```
select publicdata.climate_aggre_follow_precede
```

```
('LST night', 29.4,-1.66, 4326, 35991, '2012-02-02', '2012-06-01', 0, 7)
```

The *climate_aggre_follow_precede* function, differs from the *climate_aggre_mov* slightly. Just the following expression is added to the end part of that function. The *climate_aggre_follow_precede* function shows the results for every N days. The N is the window size.

```
WHERE m % ($8+$9+1) = 0
```

The following query is used to compute the aggregated values of *MOD11A1 LST night* data to compare with the values extracted from the *MOD11A2 LST night* data

A query is used to extract the values for the same point from the MOD11A2 LST night data:

```
SELECT date_of_data,
       max(st_value( rast, st_transform
                   ((st_GeomFromText('POINT(29.4 -1.66)', 4326)),
                    35991))) as temp1
FROM lst_night8
GROUP BY date_of_data
ORDER BY date_of_data
```

Table 5.1: Comparison of eight-daily data and eight-day aggregated data as a measure for validation

Date	eight-day data	eight-day aggregated data	Difference between two approaches
2012-02-02	14067	14067.33333	-0.333333333
2012-02-10	14021	14020.5	0.5
2012-02-18	14098	14098	0
2012-02-26	14051	14051	0
2012-03-05	14139	14138.66667	0.333333333
2012-03-13	13982	13982	0
2012-03-21	14093	14093.33333	-0.333333333
2012-03-29	14015	14015	0
2012-04-06			
2012-04-14	14194	14194	0
2012-04-22	14014	14014.33333	-0.333333333
2012-04-30			
2012-05-08	14083	14083	0
2012-05-16	13890	13890	0
2012-05-24	14102	14102	0
2012-06-01	14078	14077.8	0.2

Part of the results to compare the extracted values from the two approaches are in Table 5.1. The difference in the results from the two data sources does not exceed more than 0.5. The difference is most probably due to rounding the MOD11A2 eight-day data to the nearest integer. By this assumption, the aggregated values from the MOD11A1 are even more precise. By rounding these values to the nearest integer, identical values from the two approaches are derived. This test has been done for a number of points, and the same outcome has always resulted. This can be considered as a good validation measure for the aggregation. Since the test is done on the overall system and from two different products, the identical values result, we can consider this as a validation measure for the system.

5.4 DATA AGING

To keep the data for long periods of time, and prevent the storage from becoming too voluminous at the same time, a gradual data aggregation mechanism is needed [29]. This means that the newer data should be kept in less summarized mode than the older data. In the other words, as the data ages, higher aggregated levels must replace the original data or the finer granularities of the data. The hypothesis is also supported by [57]. After a certain interval at time, the data stream should assume to be stopped and then the data in the database should start undergoing a gradual aggregation based on certain rules. This is done for maintaining a balance between the usability of the data as the data gets older and the data volume [49].

5.4.1 Proposed mechanism

The aggregation level depends on the age of data. Defining a set of standards and rules for the level of aggregation for older data is beyond the scope of this research and can be the subject of future work, but to our knowledge there is no set of standards for aggregation level in agriculture. Here, we define an example of such rules to explain the methodology. No aggregation function is imposed over the newest data, but the data can be aggregated to higher levels of aggregation as long as the summarized information is meaningful. The data can be removed from the system if no use is expected on the horizon. For one thing, coffee growing is characterized by a biennial production cycle. This means the conditions of final product can be affected from the climate conditions over the last two years. This triggered us not to aggregate the data that is younger than two years old. Setting such rules needs more investigation and study. The following example for granularity-based data aggregation rules can be defined in the context of this project:

1. When data is 2 years old aggregate to 10 day level.
2. When data is 4 years old aggregate to 20 day level.
3. When data is 8 years old aggregate to 40 day level.

Iftikhar [29] proposed an algorithm to apply every rule, insert the aggregated data to the data store, delete all the rows from the data store that have been aggregated, and then go for further aggregation (the next rule). Since the aggregation is a costly operation for our application, we try to aggregate all the levels simultaneously. In the above-mentioned example, this means that for the data younger than two years old, the database remains intact, for the data between two and four years old 10 day aggregation, for the data between four and eight years old 20 day aggregation, and for the data older than eight years 40 day aggregation will be applied.

Applying the algorithm proposed by Iftikhar [29] causes some data to be aggregated more than once. For example, firstly it is aggregated to 10 day data and then data older than four years, aggregated further to 20 day data. In our proposed algorithm, all the data are aggregated to a relevant level based on their age at once. Imagine the rules has been run for the first time on the database. There should be a table in the database to register the dates of the original data, 10 day aggregate data and so on. For the second time, it is the most convenient to run the aggregation rules after 40 days (coarsest granularity level). This means that most of the records would remain intact, 40 records of daily data would be aggregated to four records of 10 day data, and another four 10 day data would be aggregated to two 20 day data. This is what happens for the data that is almost two years old. In fact, when data aging mechanism runs after the first time, only the data around two, four, and eight years old are affected. For example, the data around three years old that are already aggregated should remain intact. That is why it is important to keep track of temporal aggregation level.

Aggregation for data aging is quite different from the aggregation that is explained above to acquire summarized information for a point. The concept is the same, but in this case a number of raster files should be aggregated to a single file. Since the aggregation would be performed on every cell, not just on a point in the study area a more costly operation is expected. It is important to notice that some of the raster data that we are working with has many missing pixels that are null values. These null values should not be involved in the computations. That is why the number of *not null* values should also be recorded in the aggregated data. For example, when we want to compute 20 day data from two 10 day data, a simple averaging over the corresponding pixels would not be accurate enough and a weighted averaging (considering the number of *not null* values in each 10 day data) should be performed. The information about the number of recored (not null) values can be stored in a separate band of the same aggregated raster data.

The scripts to create an aggregated data from two other raster files outside the database is in Appendix G. But as long as all the data is available inside the database, it is much better to do all the operations inside the database. The aggregated data either can be stored in the same table as the data or in a separate table. If we can store all the aggregate data of a particular climatic parameter in the same table and then delete the original data it would be more efficient.

A database schema is suggested to support the data produced by aging mechanism. A table is added to the existing database schema to keep track of the data that has been aggregated. All the aging machine needs is that what information is aggregated to what level. If last date for each aggregation level is extracted and updated each time, all would be known for data aging mechanism. Figure 5.2 shows the updated database schema.

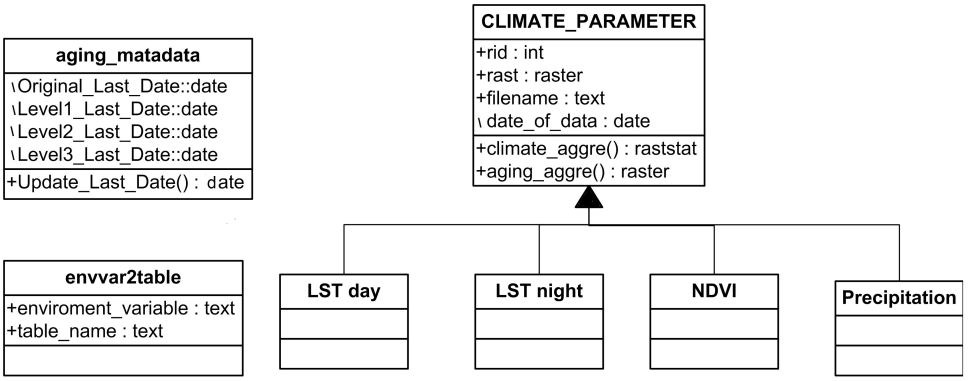

Figure 5.2: The database schema to support a data aging mechanism

The *aging_metadata* contains the last updated data in different aggregation levels. The function *Update_Last_Date()* responsibility is update the dates that are stored in the attributes of this table after every aging operation. This information makes it possible to perform different analysis on the tables with the knowledge of what record is aggregated to what level. This also makes the subsequent aging operation possible. The function *aging_aggre()* is operated on a number of raster data inside the database to produce a single aggregated raster data with corresponding information about the number of *not null* values.

The idea of all the aggregated data for every environmental parameter in one table seems feasible. It is because the raster files spatial reference system is the same, and all the information about them is kept in separate table. This issue needs more investigation and perhaps some experiments.

Chapter 6
Discussion, conclusion, and recommendation

Throughout this chapter, the different parts of the framework are scrutinized. The strong and weak points of the system are discussed, and where needed, some other methods are recommended for future studies.

6.1 DATA ACQUISITION AND DATA SOURCES

The system digests a number of data sources from different sources. The sources provide the user with the final products. In other words, every cell value represents one climatic or environmental parameter for a particular location on Earth. The system eventually is capable of absorbing any kind of raster data that has georeference information. The data is normally in ftp or http servers as a list of raster files.

Despite the fact that this includes a large number of products, the current system has a number of limitations. First of all, it does not take into account other data types. For example, the user may want to build a dynamic database of data from meteorological stations or data from Wireless Sensor Networks (WSNs) for a particular area. Some of these data sources provide even more directly streaming data. Their frequency is much higher and the data immediately is available as it is measured. Accordingly, it should be absorbed and processed in a timely fashion. There are a couple of problems associated with these kinds of data sources before they can be embedded in the framework.

A first problem is that these data sources may not be published as a list of files or records in http or ftp servers. Automated acquisition of this kind of data depends on the data source itself, and in some cases this may be even impossible, because sometimes it should be ordered each time the data issue updates. Accordingly, defining a generic solution for all data sources seems far from practical. Nevertheless, more investigation on data products would certainly lead to more generic solutions.

The other challenge is that our framework is designed to digest only raster data. The vector sources of interest can be data from WSNs and meteorological stations. There are two possible solutions to absorb vector data. The first is to perform interpolation techniques on the datasets to rasterize them. This type of sources that require an interpolation model, would certainly expand the scope of this project. This can be the case either for WSN data or data from meteorological stations. Density of the stations or sensors should be considered as an important factor to determine whether interpolation would lead to sensible results or not. In other words, if the stations or sensors are sparse, the interpolated values may not be reliable for a particular application. The other solution is that to design the application and database in a way that can directly support the data from every source. This means that some of the tables would store discrete points and their corresponding values. As a result, unlike the case of raster data, the values for every point in the study area are not directly retrievable. For a particular point in the study area, the interpolated values can be retrieved using the neighboring points inside the database. Since the costly operation of interpolation would not be performed by adopting the latter, it may be even a better

choice. Every choice the researcher adopts, first s/he should define a procedure for a particular data product, and then automate it. In other words, every data product calls for a rather special treatment.

The designed system, as discussed in previous chapters, checks for the last files in the data source and last imported data in the database periodically, and updates the database for the newly added data in the data source. It means that the system assumes that there is not any missing data in the database that exists in the data source, not just for the current data, but for previous date as well. With the existing design, no such case is expected in the system, but it may happen for unexpected reasons. For future development of the system, it is recommended to consider the detecting the existence of possible missing data in the database.

MODIS LST data, MPE precipitation data and eMODIS NDVI data are chosen as example data sources. For MODIS LST data, data acquisition is facilitated from ftp and http servers, for MPE precipitation data from an ftp server and eMODIS data from an http server. It means that two examples from each protocol are practiced for data acquisition. This would hopefully cover most of the data acquisition challenges for other data sources that one waits to embed in the system.

6.2 PREPROCESSING

The framework performs some preprocessing to prepare the data for inclusion into the database. There is a number of defined stages that includes all the necessary phases that every data product (at least the ones that we have chosen) needs. The preprocessing includes unpacking, format conversion, and cropping before being imported into the database. Each data product may skip one or two stages. Besides, each file may undergo each phase in a slightly different way. Since NDVI data are a number of big files that are not all required for further processing, the unnecessary files are removed from the system right after unpacking. Format conversion is just performed on LST data to extract the required subdatasets. Cropping was done for all the data products to dispose of unnecessary parts of data.

Another stage was considered earlier in this research project. It was decided to reproject all the raster files to a spatial reference system that best fits the study area before they are entered to the database. It is considered unnecessary and even a source for data quality loss after a few experiments. In this way, the entire database would have had a uniform spatial reference system. On the other hand, reprojection involves resampling of raster files, and this is a source for quality loss. To assess the resampling error in the case of this database and make more objective decision, a number of tests were performed. This is elaborated upon in section 6.2.1.

6.2.1 Assessing the effect of reprojection

The raster data that is acquired from different sources should have georeference information, so that the values for the points in the study area for the same coordinate system can be extracted. Normally, the data have georeference information, but spatial reference system for data products are different. One option that comes to mind immediately is to reproject all the data products to one uniform spatial reference system that best fits the study area. In a reprojection process, new image cells need to be resampled to establish a regular grid. In fact, the overall image is distorted and resampled to a new shape. So, new values have to be assigned to the pixels. This process would certainly involve some data quality issues. Besides, extra processing is added to the overall procedure, and this certainly makes the system slower.

Alternatively, we opted to store all the data products with their original spatial reference system. The user may want to extract information about some points in the study area. The points

spatial reference system may be different from that of the original data. The only thing that needs to be done is to do the transformation on the points that are required to be retrieved from the system. Besides, as described earlier, before the data is entered into the database, the process involves a cropping step, which needs a shapefile with the same coordinate system as the raster data. This means that for every data product a vector transformation is also needed. Point transformation inside the database is not a costly operation. Even though it should be done for all the points when extracting their values, it would not cause a major slowdown in the performance of the database. The vector transformation is done for every study area and every data product only once and can be considered as a preliminary requirement of the system before it is even started. Both point and vector transformations can be performed precisely when the georeference information for both is available.

Taking into account all the problems for both approaches, none of them seems to give important drawbacks except for the resampling errors. To assess the role of reprojecting in quality loss, and to make a more sensible decision on the options of choice, a few experiments have been performed. In the experiments, two tables are created, each one with one of the approaches for an arbitrarily selected period in time, and for LST data. The results are compared and discussed subsequently.

For a period of almost five months from 2011-11-01 to 2012-03-30, a table called *lst_day* was created that keeps the original spatial reference system of the data. For the same period in time, a table called *lst_day_4326* was created with its imported data had been the WGS84 system. The reprojection is done by adding a target spatial reference system option in *gdalwarp*. By default, it reprojects it using a nearest neighbor resampling method, which is the fastest algorithm, but with the worst interpolation quality. In fact, a simple *-t_srs EPSG:4326* option is added to the *gdalwarp* command when cropping the study area. The option includes different parts:

```
-t_srs: represents target spatial reference set,
EPSG: (European Petroliam Survey Group) is the SRID implementation authority
4326: indicates SRID (Spatial Reference System Identifier)
```

For a number of points, the pixel values for the entire period are extracted from both tables. Imagine a point with the following geographic coordinates:

```
Latitude: -1.45
Longitude: 30.01
```

The pixel values for every day and for both tables are extracted. For extraction of the values from the *lst_day_4326* table, the following SQL command is used:

```
SELECT  date_of_data ,
        max(st_value( lst_day_4326.rast,
                    (ST_GeomFromText
                     ('POINT(30.01 -1.45)', 4326)))
                       ) as temp
FROM lst_day_4326
GROUP BY date_of_data
ORDER BY date_of_data
```

Since the data is already transformed to the WGS84 system, there is no need to transform the point. To extract the values from the *lst_day* table that contains the data with original spatial reference system, this SQL commands are use:

```
SELECT  date_of_data ,
        max(st_value
      (lst_day.rast,
        ST_transform(
              (ST_GeomFromText('POINT(30.01 -1.45)', 4326)),35991))
          ) as temp1
FROM lst_day
WHERE date_of_data > '2011-10-26' and date_of_data < '2012-03-31'
GROUP BY date_of_data
ORDER BY date_of_data
```

The SRID of MODIS sinusoidal system is *35991*, and this has been added manually to the *spatial_ref_sys* table. If the reprojection did not involve any errors, the two tables should result in the same pixel values for the same days. But the reprojection does involve errors and it is clear from the results of this test. Part of the results to compare the two approaches are in Table 6.1. The table clearly shows that there are differences in the values resulting from either of approaches. For some days there, are minor differences in the values and for some others major discrepancies occur.

Table 6.1: Comparison of the effect of resampling—different results

Date	Original SRS	EPSG: 4326 Nearest Neighbor	EPSG: 4326 Cubic	Nearest Neighbor Discrepancy	Cubic Discrepancy
2011-12-09	14745	14776	14741	-31	4
2011-12-10					
2012-11-11	14605		14605	14605	0
2012-11-12	14648	14680	14658	-32	-10
2011-11-13					
2011-11-14					
2011-11-15					
2011-11-16		14765	14751	-14765	14751
2011-11-17					
2011-11-18	14718	14886	14744	-168	-26
2011-11-19					
2011-11-20	14715	14808	14732	-93	-17
2011-11-21					
2011-11-22					
2011-11-23					
2011-11-24					
2011-11-25	15053	15033	15054	20	-1

A number of points has been tested to make sure if the same results happens for all points or not. For most of the points, similar results are found. For a few points the values extracted from both approaches were identical. This happened for example for the point with the following coordinates:

```
Latitude: -1.6
Longitude: 29.4
```

The sample results can be found in Table 6.2. This probably means that some of the points fall in the image where resampling does not affect their values. But it does not prove that resampling in our case is unimportant. Conversely, it proves the inconsistency in error distribution that should be avoided. Since two different approaches resulted in identical values for some points, and similar values for others, the results also can be interpreted as a proof of validity of the approaches.

Table 6.2: Comparison of the effect of resampling—identical results

Date	Original SRS	EPSG: 4326 Nearest Neighbor
2011-01-14	14568	14568
2011-01-15		
2011-01-16		
2011-01-17	14493	14493
2011-01-18		
2011-01-19	14547	14547
2011-01-20		
2011-01-21	14780	14780
2011-01-22	14798	14798

As discussed in the previous chapter, the aggregated values from daily data are the same as the eight-day LST product of MODIS (MOD11A2). This was used as a measure for validation of the aggregation results. Trusting the algorithm that is used for computation of MOD11A2, it can be used for assessment of the errors involved in the reprojection process. Here, the eight-day aggregated values from reprojected LST files, LST file in their original spatial reference system, and extracted from eight-day product are compared in Table 6.3. The reprojection is done onto the WGS84 projection system (EPSG:4326).

Table 6.3: Comparison of the effect of resampling on aggregated values

Date	MOD11A2	eight-day aggregated original SRS	eight-day aggregated reprojected
2012-01-01	14843	14843	14830.2
2012-01-09	14754	14754	14546.34
2012-01-17	14737	14737	14812
2012-01-25	15058	157057.67	15131
2012-02-02	15033	15033	15104.34
2012-02-10	15089	15088.67	15025
2012-02-18			
2012-02-26	14492	14492.33	14506.5
2012-03-05	15091	15091	15128.67

This table clearly shows the identical (not considering the rounding error) results extracted from 8-daily data and 8-day aggregated data when there is no reprojection involved in the processes. The results are quite different when the aggregation is performed over reprojected data. All the tests in this section suggests that reprojection leads to less precise results. Besides, reprojection before the data can be entered into the database is an extra operation that can be avoided.

6.3 IMPORTING DATA INTO THE DATABASE

The system automatically imports the raster files into the database, using *PostGIS Raster*. To store all the data in the data source, in the database and have information about every point in the study area, all the pixel values and their corresponding locations in a specific coordinate system should be kept in the database. Raster as data source is considered more appropriate for this project, since climatic information about the whole study area is in demand, not just discrete points on a limited number of stations throughout the study area.

In this research project, since the data source is raster files, primarily *raster type* was chosen as data structure in the database. The extension *PostGIS Raster* supports *raster* data type. A *vector design* is considered as an alternative to raster design, that more elaborate explanation and comparison to raster design is in section 6.3.1.

6.3.1 Data structure choice

Since the storage of raster files may not be efficient enough in large databases, we considered another design with different architecture. In this design, which we call *vector design*, every corresponding pixel of the raster files holds values from all the files in an array. Such a table was built for one of the parameters (LST) and for the year 2002 with the following columns:

- **ID:** is an integer that is included automatically,
- **pix_centroid:** is a point geometry that holds the coordinates of every pixel centroid, and
- **Array_366:** An integer array of 1×366 that holds daily pixel values for an entire year.

To build such a table, first a raster table with the same data is built, and then pixel values and centroids are extracted from it. Some tests need to be done to assess the performance, data volume, and simplicity of query code and database structure for the two approaches. Before the tests, some issues needs to be considered that help make a final decision:

- The vector design is built based on the raster design and this means that first the raster design should be built and then the vector design is built on top of that. This will certainly add to the complexity of the construction of the database.
- In vector design, each pixel explicitly holds all the information for a time series from beginning to the last updated data. This can be considered as a part of results the user is looking for. This does not happen in a raster structure and calculation is needed to extract such results.
- In raster design, adding newly updated data means adding new records for each table. In vector design, the number of records is always the same for a particular parameter. The number of records is equal to the number of pixels in the raster files. Even the number of attributes is the same. Only the size of the array is subject to change. Updating the table in this case leads to a larger array for all the records. This cannot be considered a major advantage or disadvantage and just shows the distinction between the two architectures that should be taken into consideration.

To compare the size of the tables and other tests, two tables were built each for either approach for the entire year 2002. To determine the size of the each table, this query code was used:

```
SELECT pg_size_pretty(pg_total_relation_size('"my_schema"."my_table"'));
```

The disk size of the raster table is 9.4 MB, and 110 MB for vector design. We decided to find the actual filenames for the tables and verify their size on disk at file system level. This piece of SQL commands lists all the filenames of tables for a given schema. Here, the schema name is *ashouri*:

```
SELECT t.relname, current_setting
                  ('data_directory')||'/'||pg_relation_filepath(t.oid)
FROM pg_class t
    JOIN pg_namespace ns on ns.oid = t.relnamespace
WHERE relkind = 'r' and ns.nspname = 'ashouri';
```

By checking the filenames and paths that this query provides, the size of raster table is less than 1 MB and the size of vector table is 93 MB. The storage characteristics clearly support the raster-based solution. But it is not enough to make a decision, and more tests are needed. Nowadays, the performance of the system is more important than storage. The problem of storage should be managed by adopting a robust data aging mechanism in streaming data management systems. The simplicity of the query code is also important for us because simpler code leads to better system maintainability. First, the performance of a couple of queries against the data on either side is scrutinized.

Imagine we need to extract the pixel value on a specific date. We used EXPLAIN ANALYSE command to calculate the total runtime. Query for raster design:

```
SELECT max(st_value
           (rast,st_transform(
                 (st_GeomFromText
                 ('POINT(30.424 -1.66)', 4326)), 35991)))
                 *0.02-273.15 as temp1
FROM rwanda_sin1
WHERE date_of_data='2002-01-09'

total runtime: 2.047 ms
```

For the vector design, first a spatial *GIST index* is created and then this query is built:

```
WITH B as(
SELECT array_366
FROM pixelbased3
WHERE ST_Intersects
      (ST_SetSrid(ST_MakeBox2D
                 (ST_GeomFromText
                 ('POINT(3382041.509 -185046.786)'),
                  ST_GeomFromText('POINT(3381115.509 -184120.786)')),35991),
                  pix_centroid)
)
SELECT (array_366[(select EXTRACT (doy FROM TIMESTAMP '2001-01-09'))])
       *0.02-273.15
FROM B

total runtime: 0.72 ms
```

A *bbox* is constructed around the query point of same size of pixels, and simply intersected with pixel centroids. The query eventually finds one relevant pixel.

This query is faster than all the others and this means for extracting a single point value from the database, we reached a faster solution in the vector design.

To finalize the choice, another set of commands that computes the aggregated values for a single point is compared for each approach.

In this example, it is required to compute the six-day aggregated data for the year 2002 and for the same point from both tables. The query for raster design was already explained in the previous chapter. The total runtime of that query for the same point, for the year 2002 and six-day aggregated data is 561.831 ms. For the vector design, since any of the arrays should be partitioned into the groups of six, a function is defined to aggregate and automate the job for all the arrays.

```
CREATE OR REPLACE FUNCTION ashouri.array366_aggregate
                           (p_array_366 numeric [])
RETURNS numeric [] AS
$body$
DECLARE
  dim_start int = array_length(p_array_366, 1); --size of input array
  dim_end int = 61; -- size of output array
  dim_step int = dim_start / dim_end; --avg size
  tmp_sum NUMERIC; --sum of the batch
  result_array NUMERIC[61]; -- resulting array
BEGIN
  FOR i IN 1..dim_end LOOP --from 1 to 61.
    tmp_sum = 0;

    FOR j IN (1+(i-1)*dim_step)..i*dim_step LOOP --from 1 to 6, 7 to 12, ...
      tmp_sum = tmp_sum + p_array_366[j];
    END LOOP;

    result_array[i] = tmp_sum / dim_step;
  END LOOP;

  RETURN result_array;
END;
$body$
```

The aggregated data can be calculated with the following query, making use of the defined function:

```
SELECT id, array366_aggregate(array_366)
FROM pixelbased3
WHERE ST_Intersects(
            ST_SetSrid(ST_MakeBox2D
            (ST_GeomFromText('POINT(3382041.509 -185046.786)'),
               ST_GeomFromText('POINT(3381115.509 -184120.786)')),35991),
            pix_centroid)
```

The total runtime of this query is 746.302 ms. Despite the fact that vector design for aggregated data seem to be slower than the raster design (at least with this attempt to aggregate the data), it cannot be considered as a major drawback. The reason is that the total runtime of both queries did not exceed more than one second and both seem to be fast enough. The complexity of the vector design is on a par with the raster design or even simpler. Both approaches have merits and demerits that have been explored in this section. The final decision depends on the requirements of the user and database designer. We decided to choose the raster design, because at least for the case of this study, the advantages of this design seem to outweigh the disadvantages.

6.3.2 Database design

We have defined a table for every environmental variable to digest the data and designed a database based on that. Every table digests the data from a separate data source. But one may consider the possibility of putting all the data from different sources in one table. Putting all the raster files in just one column does not seem to be a good idea. The extraction of relevant information from such a table would be troublesome. For one thing, the spatial reference system of every product is different and a query to extract information would become complicated or impossible.

If we put every data product in separate column of a table, this can be possible. One should consider that the different data products do not issue updates simultaneously. Besides, their frequency may not be the same. Furthermore, for some of the data sources more than one raster file is needed. So, their corresponding identifier or date column cannot be the same. Needless to say, their filenames cannot be the same as well. Accordingly, every parameter needs at least one accompanying column to store the date of data, and another column to store filenames. Considering these conditions, storing all the environmental variables inside one big table is possible theoretically.

The *raster2pgsql* loader imports the data from outside the database, and in our case we made it work automatically. It creates table and three columns by default and start populating the table. This technical issue makes the problem challenging, but it does not seem an impossible hurdle to clear. For every data product a separate data loading command should be run. Every loader creates a new table. But the data can be appended to an existing table and this would not cause an important problem. The other problem is that every *raster2pgsql* command creates an *rid* column by default as its primary key. Multiple primary keys are not allowed in one table. We have not investigated thoroughly the possibility of resolving this issue, but at least *raster2pgsql* options do not give any option about dropping the primary key constraint. If one meets all the challenges, it is probably a good option. But our design does not have a serious disadvantage in comparison to this design. Besides, it does not have the technical challenges. It is worth to mention that we have not implemented a database with just one table to store the entire data products and this design deserves more research.

6.4 AUTOMATED SYSTEM

The system checks for updates every few hours and populates the database with preprocessed and timely data. The automated system is expected to be run flawlessly forever, but this expectation is far from practical. This is mainly because the functionality of the overall system depends on a number of factors such as functionality of the data source servers, the Internet connection, and performance of the ITC server. An error handling mechanism by e-mail was designed and it is expected that it reports most of the unexpected errors in the system. The system itself is considered to be fully automated, but cannot deliver a generic solution when a different data source is required to be embedded as a part of the system. Nevertheless, with a moderate programming

understanding, it can be adapted to every continuously updated data source. The framework can be deployed to every other server with minor changes in the scripts. It is recommended to add an interface to the scripts to control the variables when it is needed.

To include an extra environmental variable a number of steps should be followed:

- If the data source is in ftp or http servers as a list of continuously updated data, one of the functions that is defined for data acquisition is useful. It should be noticed that the date of data should be extracted from the filename or the folder name. The functions should be slightly adapted with the new data sources. If the files are accessible in a network, like the environmental data that are available with their updates in ITC network, The procedure is even easier. First the frequency of the data updates should be determined. The last updated file can be considered as new data that should be imported into the database. All the other files are ignored. There is no need to download or copy the data if it is readily accessible by the network. For the cases that the data has to be ordered or has to be accessed using an application, data acquisition methods offered in this project, would not help much and for every case a different procedure should be defined.

- The format, spatial reference system, and size of the acquired data should be determined. If the data is compressed file, an uncompressing stage is needed as a part of prepossessing. After uncompressing the files only some (or one) of the uncompressed files files may be useful that should be involved in the processes. Other files either are dependent files to the main file, or contain other information (that may not be of interest).

  As discussed earlier, GDAL supports many raster file formats and in the scripts we assumed that the format of input files is supported by GDAL. Just in case the format of the new data source is not supported, there are two possible solutions. First is to provide an automatic format conversion facility. The conversion should be done into one of GDAL supported formats, and after that procedures such as cropping and importing into the database can be followed like the ones in this project. If the conversion is not possible, since cropping and importing into the database is done for GDAL supported formats, different approach should be taken and the methods and functions of this project would not be useful.

  In post-processing stage, the date of data should be extracted. In the case of our files it is embedded in the filename. It may not be the case for all the data sources. The date of data should be part of metadata or be the same as the data acquisition date. This date should be extracted and put a separate column in the database, because it is needed in the subsequent processes over the database.

6.4.1 Distribution of functionality between Python scripts and the database

The automated system, operates on files that are in remote data sources. It connects to the server, acquires the data that are not existed in the database, do some operations on it to make it ready to enter into the database, and do some post-processing over the data inside the database. All the process is done inside the Python script, but it connects to the database three times during the automated process. It connects to the database using *DAL*, and runs a small query to determine the last updated record in the database. It is important as a part of automated process to determine which file to download. It also connects to a table inside the database using *raster2pgsql* to populated the database with the downloaded and processed data. It connects once more to the database fill the *date_of_data* column with derived date from the corresponding filename. Finally, it connects to the database to fill the missing dates. These processes are done inside the Python scripts, but actually makes use of PostgreSQL and the tables inside the database to do that. The

idea behind calling the SQL commands from the application was to involve them in the automated process.

6.5 AGGREGATION AND DATA AGING

We tried to address most of the information requirements of the user that can be derived from this framework, using aggregation methods. A function was defined that calculates aggregated data with the required aggregation level, for a specific point, and specific time period. Another function calculates the aggregated values for a specific time of the year for multiple years. A moving window function is defined to calculate the aggregated data in the following and preceding days. This can be considered as a smoothing mechanism and also the results can directly be useful for the user. A research on suitability of each approach, and also the optimal granularity for the application is needed as a follow-up to this project. Besides, spatial smoothing techniques (such as the ones that are used in remote sensing) is recommended to be studied and implemented on the image data to handle the issue of missing pixels.

It is imperative for a streaming data management system to manage large amounts of data. The management of outdated or less important data should be part of such a system. Throughout this thesis, just a theoretical foundation of such a mechanism for our system is discussed and a methodology is proposed. Full implementation of this proposal would present new challenges to the researcher.

6.6 CONCLUSION

The primary objective of this research was to build a continuously updated spatio-temporal database for climatic data. A number of subobjectives are followed to build such a system. First, we tried to make a prototype system for each parameter. Following the procedures to build a prototype system, invaluable information were acquired on how to construct the final framework. In fact, each step that should be taken in the automated system automatically was performed manually in our prototype system. We gathered information on how to acquire the data, and how to manipulate the data and make it ready to be imported to a database. We tried to make a connection between our application and the DBMS, and load the preprocessed data from within the application as a part of automated procedure.

Through a number of tests we proved that reprojection of the raster data to a single spatial reference system before entering into the database leads to less precise results. It is recommended that the raster data should keep their original spatial reference system in the database.

The extension *PostGIS Raster* offers facilities to store and analyze raster data within the database. The data structure of the records in the database would be raster. A number of tests on the choice of data structure was performed to demonstrate that this data structure is a good option for a dynamic data management facility in terms of performance and data volume.

The outcome of this research project is a database of climatic information that is updated as the data in the data source is updated. Since the data source is in raster format, it is expected that the system delivers a continuous database, for each point in the study area. It does not happen in practice for all the parameters because of missing pixels in the files and missing data in the data source archive. Calculation of higher aggregated data is facilitated as a part of this system. This provides the user with summarized information. Besides, more aggregated data has less missing values. Additionally, for the problem of missing pixels, a moving average filter has been defined to take into account the preceding and following measured data in the same cell. This fills some of the missing pixels with approximate values. The performance of the system is satisfactory and

is a good example of continuously updated climatic database for future development and analysis.

With the experience gained from this project, we would do the next project in a slightly different way. For instance, I came up with the design and the overall procedure in the last stages of this project. Part of this is inevitable, but a general sketch of design and procedure would certainly help to take more sensible steps in each step of the project. More processes inside the database would have been desirable, since the data is already entered into the database and also many spatial operations has been facilitated through spatial extension of DBMSs.

Bibliography

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] R. Adler, G. Huffman, A. Chang, R. Ferraro, P. Xie, J. Janowiak, B. Rudolf, U. Schneider, S. Curtis, D. Bolvin, et al. The Version-2 Global Precipitation Plimatology Project (GPCP) monthly precipitation analysis (1979-present). *Journal of Hydrometeorology*, 4(6):1147–1167, 2003.

[3] A. Arasu, S. Babu, and J. Widom. CQL: A language for continuous queries over streams and relations. In *Database Programming Languages*, pages 123–124. Springer, 2004.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[5] S. Babu and J. Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3): 109–120, 2001.

[6] I. Becker-Reshef, C. Justice, M. Sullivan, E. Vermote, C. Tucker, A. Anyamba, J. Small, E. Pak, E. Masuoka, J. Schmaltz, et al. Monitoring global croplands with coarse resolution earth observations: The Global Agriculture Monitoring (GLAM) project. *Remote Sensing*, 2(6):1589–1609, 2010.

[7] L. Cabibbo and R. Torlone. A framework for the investigation of aggregate functions in database queries. *Database Theory–ICDT'99*, pages 383–397, 1999.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. Shah. TelegraphCQ: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668. ACM, 2003.

[9] S. Chun, C. Chung, J. Lee, and S. Lee. Dynamic update cube for range-sum queries. In *Proceedings of the International Conference on Very Large Data Bases*, pages 521–530, 2001.

[10] A. Colombi, C. De Michele, M. Pepe, and A. Rampini. Estimation of daily mean air temperature from MODIS LST in Alpine areas. *EARSeL eProceedings*, 6(1):38–46, 2007.

[11] C. Daly. Guidelines for assessing the suitability of spatial climate data sets. *International Journal of Climatology*, 26(6):707–721, 2006.

[12] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proceedings of the thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 635–644. Society for Industrial and Applied Mathematics, 2002.

[13] M. DiPierro. *Web2py Enterprise Web Framework*. Wiley Publishing, 2009.

[14] K. Douglas and S. Douglas. *PostgreSQL: A comprehensive guide to building, programming, and administering PostgreSQL databases*. SAMS publishing, 2003.

[15] A. Downey. *Python for software design: how to think like a computer scientist*. Cambridge University Press, 2009.

[16] M. Feng, C. Huang, S. Channan, E. Vermote, J. Masek, and J. Townshend. Quality assessment of Landsat surface reflectance products using MODIS data. *Computers & Geosciences*, 38(1):9–22, 2012.

[17] J. Garratt. *The atmospheric boundary layer*. Cambridge university press, 1994.

[18] W. Gibson, C. Daly, T. Kittel, D. Nychka, C. Johns, N. Rosenbloom, A. McNab, and G. Taylor. Development of a 103-Year High-Resolution Climate Data Set for the Conterminous United States. 2002.

[19] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[20] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. 1995.

[21] A. Gutiérrez and P. Baumann. Computing aggregate queries in raster image databases using pre-aggregated data. In *Proc. ICCSA*, 2008.

[22] A. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[23] T. Heinemann, A. Latanzio, and F. Roveda. The EUMETSAT Multi-sensor Precipitation Estimate (MPE). In *Second International Precipitation Working group (IPWG) Meeting*, 2002.

[24] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. *ACM SIGMOD Record*, 26(2): 171–182, 1997.

[25] T. Hertel, N. Diffenbaugh, and N. Villoria. A global, spatially-explicit, open-source data base for analysis of agriculture, forestry, and the environment: Proposal and institutional considerations 12. 2010.

[26] R. Hijmans, S. Cameron, J. Parra, P. Jones, and A. Jarvis. Very high resolution interpolated climate surfaces for global land areas. *International Journal of Climatology*, 25(15):1965–1978, 2005.

[27] K. Hornsby and M. Egenhofer. Shifts in detail through temporal zooming. In *Tenth International Workshop on Database and Expert Systems Applications, 1999. Proceedings.*, pages 487–491. IEEE, 1999.

[28] G. Huffman, R. Adler, P. Arkin, A. Chang, R. Ferraro, A. Gruber, J. Janowiak, A. McNab, B. Rudolf, and U. Schneider. The Global Precipitation Climatology Project (GPCP) combined precipitation dataset. *Bulletin of the American Meteorological Society*, 78(1):5–20, 1997.

[29] N. Iftikhar. Integration, aggregation and exchange of farming device data: A high level perspective. In *Applications of Digital Information and Web Technologies, 2009. ICADIWT'09. Second International Conference on the*, pages 14–19. IEEE, 2009.

[30] S. Kawashima, T. Ishida, M. Minomura, and T. Miwa. Relations between surface temperature and air temperature on a local scale during winter nights. *Journal of Applied Meteorology*, 39(9):1570–1579, 2000.

[31] S. Kazemitabar, U. Demiryurek, M. Ali, A. Akdogan, and C. Shahabi. Geospatial stream query processing using Microsoft SQL Server Streaminsight. *Proceedings of the VLDB Endowment*, 3(1-2):1537–1540, 2010.

[32] V. Khatri, S. Ram, R. Snodgrass, and G. O'brien. Supporting user-defined granularities in a spatiotemporal conceptual model. *Annals of Mathematics and Artificial Intelligence*, 36(1): 195–232, 2002.

[33] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982.

[34] T. Lillesand, R. Kiefer, J. Chipman, et al. *Remote Sensing and image interpretation.* Number Ed. 5. John Wiley & Sons Ltd, 2004.

[35] I. Lopez, R. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 17(2):271–286, 2005.

[36] B. Maathuis. *In Situ and Online Data Toolbox Installation, Configuration and User Guide*, volume XML Version 1.0. Faculty of Geo-Information Science and Earth Observation (ITC), 2012.

[37] E. Meijer. The world according to LINQ. *Communications of the ACM*, 54(10):45–51, 2011.

[38] T. Mitchell and P. Jones. An improved method of constructing a database of monthly climate observations and associated high-resolution grids. *International Journal of Climatology*, 25(6):693–712, 2005.

[39] M. Mokbel and W. Aref. SOLE: Scalable On-Line Execution of continuous queries on spatio-temporal data streams. *The VLDB Journal*, 17(5):971–995, 2008.

[40] M. Mokbel, X. Xiong, M. Hammad, and W. Aref. Continuous query processing of spatio-temporal data streams in place. *Geoinformatica*, 9(4):343–365, 2005.

[41] G. Mostovoy, R. King, K. Reddy, V. Kakani, and M. Filippova. Statistical estimation of daily maximum and minimum air temperatures from modis lst data over the state of Mississippi. *GIScience & Remote Sensing*, 43(1):78–110, 2006.

[42] R. Obe and L. Hsu. *PostGIS in action*. Manning Publications Co., 2011.

[43] L. Prihodko and S. Goward. Estimation of air temperature from remotely sensed surface observations. *Remote Sensing of Environment*, 60(3):335–346, 1997.

[44] L. Qiao and D. El Abbadi. RHist: Adaptive summarization over continuous data streams. 2002.

[45] P. Ramsey, S. Santilli, C. Hodgson, J. Lounsbury, and D. Blasby. Manual PostGIS, 2009.

[46] U. Schneider, T. Fuchs, A. Meyer-Christoffer, and B. Rudolf. Global precipitation analysis products of the GPCC. *Deutscher Wetterdienst*, 12, 2008.

[47] C. Sirish, C. Owen, D. Amol, H. Wei, K. Sailesh, M. Samuel, R. Vijayshankar, and R. Frederick. TelegraphCQ: Continuous dataflow processing for an uncertain world. CIDR, 2003.

[48] J. Skyt and C. Jensen. Persistent views - a mechanism for managing ageing data. *The Computer Journal*, 45(5):481—493, 2002.

[49] J. Skyt, C. Jensen, and T. Pedersen. Specification-based data reduction in dimensional data warehouses. *Information Systems*, 33(1):36–63, 2008.

[50] J. Smith and D. Smith. Database abstractions: aggregation. *Communications of the ACM*, 20 (6):405–413, 1977.

[51] C. Vancutsem, P. Ceccato, T. Dinku, and S. Connor. Evaluation of MODIS land surface temperature data to estimate air temperature in different ecosystems over Africa. *Remote Sensing of Environment*, 114(2):449–465, 2010.

[52] Z. Wan. MODIS land surface temperature products users' guide. *Institute for Computational Earth System Science, University of California, Santa Barbara, CA. http://www.icess.ucsb.edu/modis/LstUsrGuide/usrguide. html*, 2006.

[53] Z. Wan and J. Dozier. A Generalized split-window algorithm for retrieving land-surface temperature from space. *Geoscience and Remote Sensing, IEEE Transactions on*, 34(4):892–905, 1996.

[54] E. Westra. *Python Geospatial Development*. Packt Publishing, 2010.

[55] W. Yan, P. Larson, et al. Eager aggregation and lazy aggregation. In *Proceedings of the International Conference on Very Large Data Bases*, pages 345–357. Institute of Electrical and Electronics Engineers (IEEE), 1995.

[56] D. Zhang, D. Gunopulos, V. Tsotras, and B. Seeger. Temporal and spatio-temporal aggregations over data streams using multiple time granularities. *Information Systems*, 28(1):61–84, 2003.

[57] J. Zhang, M. Gertz, and D. Aksoy. Spatio-temporal aggregates over raster image data. In *Proceedings of the 12th annual ACM International Workshop on Geographic Information Systems*, pages 39–46. ACM, 2004.

Appendix A

Automated download program from ftp

```
#/usr/bin/env python
_Author = 'Farzin Ashouri'
import os
import datetime, calendar
import ftplib
class downloadFTP(object):
    '''downloads from ftp'''
    def yj_to_ymd(self, year, doy):
        """Return date as e.g. '2010.11.25' based on specified year and
        numeric day-of-year (doy) """
        date = datetime.datetime.strptime('%d%03d' % (year, doy), '%Y%j')
        print date.strftime('%Y.%m.%d')
        return date.strftime('%Y.%m.%d')


    def get_doy_range(self, year, month):
        """Determine starting and ending numeric day-of-year (doy)
        asscociated with the specified month and year.

        Arguments:
        year -- four-digit integer year
        month -- integer month (1-12)

        Returns tuple of start and end doy for that month/year.
        """
        last_day_of_month = calendar.monthrange(self.year, self.month)[1]
        start_doy = int(datetime.datetime.strptime('%d.%02d.%02d' % (year,
            month, 1), '%Y.%m.%d').strftime('%j'))
        end_doy = int(datetime.datetime.strptime('%d.%02d.%02d' % (year,
            self.month, last_day_of_month), '%Y.%m.%d').strftime('%j'))
        return (int(start_doy), int(end_doy))

    def list_contents(self, ftp):
        """Parse ftp directory listing into list of names of the files
        and/or directories contained therein. """
        listing = []
        ftp.dir(listing.append)
        contents = [item.split()[-1] for item in listing[1:]]
        return contents
```

```
def download_mod11a1(self, destdir, tile, start_doy, end_doy, year, ver=5):
    """Download into destination directory the MODIS 11A2 HDF files
    matching a given tile, year, and day range. If for some unexpected
    reason there are multiple matches for a given day, only the first is
    used. If no matches, the day is skipped with a warning message.

    Arguments:
    destdir -- path to local destination directory
    tile -- tile identifier (e.g., 'h08v04')
    start_doy -- integer start of day range (0-366)
    end_doy -- integer end of day range (0-366)
    year -- integer year (>=2000)
    ver -- MODIS version (4 or 5)

    Returns list of absolute paths to the downloaded HDF files.
    """
    # connect to data pool and change to the right directory
    ftp = ftplib.FTP('e4ftl01.cr.usgs.gov')
    ftp.login('anonymous', '')
    ftp.cwd('MOLT/MOD11A1.%03d' % ver)
    ftp1 = ftplib.FTP('e4ftl01.cr.usgs.gov')
    ftp1.login('anonymous', '')
    ftp1.cwd('MOLT/MOD11A1.%03d' % ver)
    # make list of daily directories to traverse
    df= downloadFTP()
    available_days = df.list_contents(ftp)
    desired_days = [df.yj_to_ymd(year, x) for x in range(start_doy, end_doy+1)]
    days_to_get = filter(lambda day: day in desired_days, available_days)
    if len(days_to_get) < len(desired_days):
        missing_days = [day for day in desired_days if day not in days_to_get]
        print 'skipping %d day(s) not found on server:' % len(missing_days)
        print '\n'.join(missing_days)
    # get each tile in turn
    hdfs = list()
    print 'days=', days_to_get
    for day in days_to_get:
        ftp.cwd(day)
        files_to_get = [file for file in df.list_contents(ftp)
            if tile in file and file[-3:]=='hdf']

        if len(files_to_get)==0:
            ftp = ftplib.FTP('e4ftl01.cr.usgs.gov')
            ftp.login('anonymous', '')
            ftp.cwd('MOLT/MOD11A1.%03d' % ver)


        if len(files_to_get)==0:
```

```
            # no file found -- print message and move on
            print 'no hdf found on server for tile', tile, 'on', day
            continue
        elif 1<len(files_to_get):
            # multiple files found! -- just use the first...
            print 'multiple hdfs found on server for tile', tile, 'on', day
        file = files_to_get[0]
        local_file = os.path.join(destdir, file)
        print local_file
        ftp.retrbinary('RETR %s' % file, open(local_file, 'wb').write)
        hdfs.append(os.path.abspath(local_file))
        ftp.cwd('..')


    # politely disconnect
    ftp.quit()
    # return list of downloaded paths
    return hdfs


def download_perc(self, destdir, start_doy, end_doy, year):
    """Download into destination directory
    """
    # connect to data pool and change to the right directory
    ftp = ftplib.FTP('ftp.itc.nl')
    ftp.login('anonymous', '')
    ftp.cwd('pub/mpe/msg')
    df= downloadFTP()
    # make list of daily directories to traverse
    available_days = df.list_contents(ftp)
    desired_days = [df.yj_to_ymd(year, x) for x in range(start_doy, end_doy+1)]
    zips = list()
    try:
        for day in desired_days:
            fmt = '%Y.%m.%d'
            dt_day = datetime.datetime.strptime(day, fmt)
            tt_day = dt_day.timetuple()
            W_day = '%s%s%s' % (day[0:4], day[5:7], day[8:10])
            file =  'summsgmpe_%s.zip' %W_day
            print 'file=', file
            local_file = os.path.join(destdir, file)
            print local_file
            #print 'lf=', local_file
            try:
                ftp.retrbinary('RETR %s' %file, open(local_file, 'wb').write)
            #ftp.retrbinary('RETR %s' % file, open(local_file, 'wb').write)
                zips.append(os.path.abspath(local_file))
            except:
```

```
                pass
        except:
            pass

        # politely disconnect
        ftp.quit()
        # return list of downloaded paths
        return zips

    def mod11a1_All(self, dir, tile, start_Year, end_Year, start_DOY, end_DOY, ver =5):
        df = downloadFTP()
        if start_Year < end_Year:
            df.download_mod11a1(dir, tile, start_DOY, 366, start_Year)
            for j in range(start_Year+1, end_Year):
                df.download_mod11a1(dir, tile, 1, 366, j)
            df.download_mod11a1(dir, tile, 1, end_DOY, end_Year)
        else:
            df.download_mod11a1(dir, tile, start_DOY, end_DOY, end_Year)


    def perc_All(self, dir, start_Year, end_Year, start_DOY, end_DOY):
        df = downloadFTP()
        if start_Year < end_Year:
            df.download_perc(dir, start_DOY, 366, start_Year)

            for j in range(start_Year+1, end_Year):
                df.download_perc(dir, 1, 366, j)

            df.download_perc(dir, 1, end_DOY, end_Year)

        else:
            df.download_perc(dir, start_DOY, end_DOY, start_Year)
```

Appendix B

Automated download program from http

```
#/usr/bin/env python
_Author = 'Farzin Ashouri'
import urllib2
import os, datetime
class download_url(object):
    ''' '''
    def NDVI_oneyear(self, dir, year, start_DOY, end_DOY):
        address = r'http://dds.cr.usgs.gov/emodis/Africa/historical/TERRA/'
        os.chdir(dir)
        for i in range(start_DOY, end_DOY):
            try:
                if i<10:
                    url = address + '%s' %year + r'/comp_00%d/' %i
                    dataset = urllib2.urlopen(url)
                    pdataset = dataset.read()
                    fn = pdataset[594:653]
                    url2 = address + '%s' %year + r'/comp_00%d/' %i + fn
                elif i<100:
                    url = address + '%s' %year + r'/comp_0%d/' %i
                    dataset = urllib2.urlopen(url)
                    pdataset = dataset.read()
                    fn = pdataset[594:653]
                    url2 = address + '%s' %year + r'/comp_0%d/' %i + fn
                else:

                    url = address + '%s' %year + r'/comp_%d/' %i
                    dataset = urllib2.urlopen(url)
                    pdataset = dataset.read()
                    fn = pdataset[594:653]

                    url2 = address + '%s' %year + r'/comp_%d/' %i + fn

                print url2
                dataset2 = urllib2.urlopen(url2)
                CHUNK = 16 * 1024
                with open(pdataset[594:653], 'wb') as dl:
                    while True:
                        peice = dataset2.read(CHUNK)
                        if not peice: break
```

```
                    dl.write(peice)
            except:
                pass

    def NDVI_All(self, dir, start_Year, end_Year, start_DOY, end_DOY):
        if start_Year < end_Year:
            self.NDVI_oneyear(dir, start_Year, start_DOY, 366)

            for j in range(start_Year+1, end_Year):
                self.NDVI_oneyear(dir, j, 1, 366)

            self.NDVI_oneyear(dir, end_Year, 1, end_DOY)
        else:
            self.NDVI_oneyear(dir, start_Year, start_DOY, end_DOY)

    def yj_to_ymd(self, year, doy):
        date1 = datetime.datetime.strptime('%d%03d' % (year, doy), '%Y%j')
        return date1

    def findnth(self, haystack, needle, n):
        parts= haystack.split(needle, n+1)
        if len(parts)<=n+1:
            return -1
        return len(haystack)-len(parts[-1])-len(needle)
    def LST_oneyear(self, tile, year, start_doy, end_doy):
        d = download_url()
        address = r'http://e4ftl01.cr.usgs.gov/MOLT/MOD11A1.005/'
        #os.chdir(dir)
        date1 = self.yj_to_ymd(year, start_doy)
        date2 = self.yj_to_ymd(year, end_doy)
        print type(date2)
        day = datetime.timedelta(days=1)

        while date1 <= date2:
            try:

                datetoget = date1.strftime('%Y.%m.%d')
                date1 = date1 + day

                url = address + datetoget
                print url
                dataset = urllib2.urlopen(url)
                pdataset = dataset.read()
                print pdataset.find(tile)
                print d.findnth(pdataset, tile, 4)
                fn = pdataset[187888:187933]
                url2 = address + datetoget +'/' +fn
                print url2
```

```
            ul = urllib2.urlopen(url2)
            CHUNK = 16 * 1024
            with open(fn, 'wb') as dl:
                while True:
                    peice = ul.read(CHUNK)
                    if not peice: break
                    dl.write(peice)
        except:
            pass


def LST_All(self, tile, start_Year, end_Year, start_DOY, end_DOY):

    if start_Year < end_Year:
        self.LST_oneyear(tile, start_Year, start_DOY, 366)
        for j in range(start_Year+1, end_Year):
            self.LST_oneyear(dir, tile, 1, 366, j)
        self.LST_oneyear(tile, end_Year, 1, end_DOY)
    else:
        self.LST_oneyear(tile, end_Year, start_DOY, end_DOY)
```

Appendix C

Automated preprocessing

```
#/usr/bin/env python
_Author = 'Farzin Ashouri'
import os
import datetime, calendar
import ftplib, zipfile, re, glob

class Tran(object):
    '''Our Main Class'''

    def __init__(self, dir, my_host, my_port,
                 my_DB, my_user, my_table, my_sql):
        '''constructor - Define global variables for using in all methods'''
        self.dir = dir
        self.my_host = my_host
        self.my_port = my_port
        self.my_DB = my_DB
        self.my_user = my_user
        self.my_table = my_table
        self.my_sql = my_sql
        self.files = self.list_of_files()
    def unpack(self):
        os.chdir(self.dir)
        Li = os.listdir(self.dir)
        print Li
        #try:
        for i in Li:
            print i
            try:
                z = zipfile.ZipFile(i, 'r')
                z.extractall(self.dir)
            except:
                pass
        #except:
            #pass


    def list_of_files(self):
        ''' Automated preproccessing '''
        #Lets use a safer way:
```

```
        os.chdir(self.dir)
        if os.path.isdir(self.dir):  # check for directory availability
            Li = os.listdir(self.dir)
            return Li
        else:
            print('Error [No 01]! Directory is not availabe.')
            sys.exit()  # We need to stop program

    def cov_format_day(self):
        '''' converts the format to GeoTiff format '''
        for i in self.files:
            f_hdf = 'gdal_translate HDF4_EOS:EOS_GRID:"%s":MODIS_Grid_Daily_1km_LST'
            ':LST_Day_1km %s_day.tif' % (i, i)
            os.system(f_hdf)
    def cov_format_night(self):
        '''' converts the format to GeoTiff format '''
        for i in self.files:
            f_hdf = 'gdal_translate HDF4_EOS:EOS_GRID:"%s":MODIS_Grid_Daily_1km_LST'
            ':LST_Night_1km %s_night.tif' % (i, i)
            os.system(f_hdf)
    def crop_area(self, study_area):
        ''' crops the data according to the study area '''
        Li=self.list_of_files()
        print Li
        for i in Li:
            os.system('gdalwarp -cutline %s -dstnodata None -crop_to_cutline'
            ' %s %s_cropped.tif'
                      % (study_area, i, i))

    def crop_area_NDVI(self, study_area):
        ''' crops the data according to the study area '''
        Li=self.list_of_files()
        print Li
        for i in Li:
            if 'VI_NDVI' in i:
                if '.tif' in i:
                    print i
                    os.system('gdalwarp -cutline %s -dstnodata None'
                    ' -crop_to_cutline %s %s_cropped.tif'
                      % (study_area, i, i))

    def R2db(self):
        ''' creates an sql file to import the rasters to the database '''
        os.system('raster2pgsql -r *_cropped.tif -a -F %s.%s > %s.sql' %
                  (self.my_user, self.my_table, self.my_sql))

    def R2db_LST(self, day_night):
        ''' creates an sql file to import the rasters to the database '''
```

```
        os.system('raster2pgsql -r *%s.tif_cropped.tif -a -F %s.%s > %s.sql' %
                    (day_night, self.my_user, self.my_table, self.my_sql))


    def import2DB(self):
        ''' imports an external sql file to the database '''
        os.system('psql -h %s -w -U %s -p %s -d %s -f %s.sql' %
             (self.my_host, self.my_user, self.my_port, self.my_DB, self.my_sql))

    def raw_hdf_2DB(self, study_area, day_night):
        ''' All the procedures will be executed here in order'''
        ##self.list_of_files()
        if day_night == 'day':
            self.cov_format_day()
        elif day_night == 'night':
            self.cov_format_night()
        self.crop_area(study_area)
        self.R2db_LST(day_night)
        self.import2DB()

    def raw_perc_2DB(self, study_area):
        ''' All the procedures will be executed here in order'''
        self.unpack()
        self.crop_area(study_area)
        self.R2db()
        self.import2DB()
    def removeFiles(self, pattern):
        files_to_remove = glob.glob(pattern)
        for i in files_to_remove:
            os.remove(i)

    def raw_NDVI_2DB(self, study_area, pattern):
        ''' All the procedures will be executed here in order'''
        self.unpack()
        self.crop_area_NDVI(study_area)
        self.R2db()
        self.import2DB()
```

Appendix D

Automated date and data management facility

```
#/usr/bin/env python
_Author = 'Farzin Ashouri'
import datetime, calendar, os
from ftp_dl import *
from dl_url import *
from tran import *
from dal import *
import urllib2
import smtplib, sys
import traceback

db=DAL('postgres://ashouri:<password>@gip.itc.nl:5433/gncagg')
class date_func():
    ''' '''
    def LastNDVI(self):
        url = r'http://dds.cr.usgs.gov/emodis/Africa/historical/TERRA/'
        dataset = urllib2.urlopen(url)
        pdataset = dataset.read()
        LastYear_NDVI = (pdataset[(pdataset.find('</html>')-82)
         : (pdataset.find('</html>')-78) ])
        url = url + LastYear_NDVI
        dataset = urllib2.urlopen(url)
        pdataset = dataset.read()
        LastDay_NDVI = (pdataset[(pdataset.find('</html>')-87)
        :(pdataset.find('</html>')-84)] )
        return (int(LastYear_NDVI), int(LastDay_NDVI))

    def LastLST(self):
        df = downloadFTP()
        ftp_LST = ftplib.FTP('e4ftl01.cr.usgs.gov')
        ftp_LST.login('anonymous', '')
        ftp_LST.cwd('MOLT/MOD11A1.005')
        S = df.list_contents(ftp_LST)[-1]
        fmt='%Y.%m.%d'
        dateLST = datetime.datetime.strptime(S, fmt)
        tt_ds_LST = dateLST.timetuple()
        last_day_ds = tt_ds_LST.tm_yday
        last_year_ds = tt_ds_LST.tm_year
        return (last_year_ds, last_day_ds)
```

```
def LastPerc(self):
    df = downloadFTP()
    ftp_perc = ftplib.FTP('ftp.itc.nl')
    ftp_perc.login('anonymous', '')
    ftp_perc.cwd('pub/mpe/msg')
    S = df.list_contents(ftp_perc)[-1]
    fmt='%Y%m%d'
    datePerc = datetime.datetime.strptime(S[10:18], fmt)
    tt_ds_Perc = datePerc.timetuple()
    last_day_ds = tt_ds_Perc.tm_yday
    last_year_ds = tt_ds_Perc.tm_year
    return (last_year_ds, last_day_ds)

def lastDB(self, tableName):
    try:
        last_date_db = db.executesql("select date_of_data "
        "from %s "
        "order by date_of_data desc "
        "limit 1 " % tableName)
        dt_db = last_date_db[0][0]
        tt_db = dt_db.timetuple()
        last_day_db = tt_db.tm_yday
        last_year_db = tt_db.tm_year
        return (last_year_db, last_day_db)
    except:
        return (2000, 1)
        #raise

def fillDate_LST(self, my_table):
    db.executesql("UPDATE %s "
    "SET    date_of_data = to_date(substr(filename, 10, 7), 'YYYYDDD') "
    " WHERE date_of_data is Null" % my_table)
    db.commit()
def fillMissing(self, my_table):
    db.executesql("INSERT INTO %s (date_of_data, rast) "
    "SELECT x.date_of_data, NULL "
    "FROM ( "
    "SELECT generate_series(min(date_of_data), max(date_of_data), '1d')'
    '::date AS date_of_data "
    "FROM   lst_day "
    ") x "
    "WHERE NOT EXISTS (SELECT 1 FROM lst_day t WHERE t.date_of_data ='
    ' x.date_of_data) " %(my_table))
    db.commit()

def fillDate_Perc(self, my_table):
    db.executesql("UPDATE %s "
```

```
        "SET    date_of_data = to_date(substr(filename, 11, 8), 'YYYYMMDD') "
        " WHERE date_of_data is Null" % my_table)
        db.commit()
    def fillDate_NDVI(self, my_table):
        db.executesql("UPDATE %s "
        "SET    date_of_data = to_date(substr(filename, 14, 12) , 'YYYY.***-DDD') "
        " WHERE date_of_data is Null" % my_table)
        db.commit()
    def send_error(self, recipient, body):
        SMTP_SERVER = 'smtp.gmail.com'
        SMTP_PORT = 587
        password = "<e-mail_password>"
        sender = 'fashouri@gmail.com'
        subject = 'There is an error in the syetem'
        headers = ["From: " + sender,
                               "Subject: " + subject,
                               "To: " + recipient,
                               "MIME-Version: 1.0",
                               "Content-Type: text/html"]
        headers = "\r\n".join(headers)
        session = smtplib.SMTP(SMTP_SERVER, SMTP_PORT)
        session.ehlo()
        session.starttls()
        session.ehlo
        session.login(sender, password)
        send_it = session.sendmail(sender, recipient, headers + "\r\n\r\n" +  body)
        session.quit()
        return send_it
```

Appendix E

Automated program

```
#/usr/bin/env python
_Author = 'Farzin Ashouri'

import datetime
import calendar
import glob
from dal import *
from ftp_dl import *
from tran import *
from date_management import *
import smtplib
import sys
import traceback
db = DAL('postgres://ashouri:<password>@gip.itc.nl:5433/gncagg')


def LST_dlAndProcess():
    file_to_dl = os.getcwd()
    print file_to_dl
    file_to_dl = os.path.join(file_to_dl, r'LST')
    os.chdir(file_to_dl)
    print file_to_dl

    if not os.path.exists(file_to_dl):
        os.makedirs(file_to_dl)
    try:
        db.executesql('CREATE TABLE "ashouri"."lst_day" '
                      '("rid" serial PRIMARY KEY,"rast" raster,'
                       '"filename" text, "date_of_data" date); ')
        db.executesql('CREATE TABLE "ashouri"."lst_night" '
                       '("rid" serial PRIMARY KEY,"rast" raster,'
                       '"filename" text, "date_of_data" date); ')
        db.commit()
    except:
        pass

    df = downloadFTP()
    DF = date_func()
```

```
print("Files Will be Downloaded and Proccessed in ", file_to_dl)
print("Last data in Database is for:", (DF.lastDB('LST_day')[0],
      DF.lastDB('LST_day')[1]))
print("Last data in Data Source is for:", (DF.LastLST()[0],
      DF.LastLST()[1]))

try:
    df.mod11a1_All(file_to_dl,
                   'h20v09',
                   DF.lastDB('LST_day')[0],
                   r,  # DF.LastLST()[0],
                   DF.lastDB('LST_day')[1] + 1,
                   DF.LastLST()[1])

    df.mod11a1_All(file_to_dl,
                   'h21v09',
                   DF.lastDB('LST_day')[0],
                   DF.LastLST()[0],
                   DF.lastDB('LST_day')[1] + 1,
                   DF.LastLST()[1])

    T_day = Tran(
        dir=file_to_dl,
        my_host='gip.itc.nl',
                my_port='5433',
                my_DB='gncagg',
                my_user='ashouri',
                my_table='LST_day',
                my_sql='LST_day')
    T_night = Tran(
        dir=file_to_dl,
        my_host='gip.itc.nl',
                my_port='5433',
                my_DB='gncagg',
                my_user='ashouri',
                my_table='LST_night',
                my_sql='LST_night')

    T_day.raw_hdf_2DB('RWA_adm0_proj_sinu.shp', 'day')
    FL_day = glob.glob("*day*")
    for f in FL_day:
        os.remove(f)
    T_night.raw_hdf_2DB('RWA_adm0_proj_sinu.shp', 'night')
    DF.fillDate_LST('LST_day')
    DF.fillDate_LST('LST_night')
    DF.fillMissing('LST_day')
    DF.fillMissing('LST_night')
except:
```

```
        full_error = traceback.format_exc()
        full_error = str(full_error)
        open('error.txt', 'w').write(full_error)
        os.system(
            "mail -s 'There is an error in the system' "
            " -I  ashouri29188@itc.nl   < error.txt")

    FL1 = glob.glob("MOD*")
    FL2 = glob.glob('*.sql')
    for f in FL1:
        os.remove(f)
    for f in FL2:
        os.remove(f)
LST_dlAndProcess()
```

Appendix F

PostGIS Raster supported formats

```
Virtual Raster
GeoTIFF
National Imagery Transmission Format
Raster Product Format TOC format
ECRG TOC format
Erdas Imagine Images (.img)
CEOS SAR Image
CEOS Image
JAXA PALSAR Product Reader (Level 1.1/1.5)
Ground-based SAR Applications Testbed File Format (.gff)
ELAS
Arc/Info Binary Grid
Arc/Info ASCII Grid
GRASS ASCII Grid
SDTS Raster
DTED Elevation Raster
Portable Network Graphics
JPEG JFIF
In Memory Raster
Japanese DEM (.mem)
Graphics Interchange Format (.gif)
Graphics Interchange Format (.gif)
Envisat Image Format
Maptech BSB Nautical Charts
X11 PixMap Format
MS Windows Device Independent Bitmap
SPOT DIMAP
AirSAR Polarimetric Image
RadarSat 2 XML Product
PCIDSK Database File
PCRaster Raster File
ILWIS Raster Map
SGI Image File Format 1.0
SRTMHGT File Format
Leveller heightfield
Terragen heightfield
USGS Astrogeology ISIS cube (Version 3)
USGS Astrogeology ISIS cube (Version 2)
NASA Planetary Data System
```

```
EarthWatch .TIL
ERMapper .ers Labelled
NOAA Polar Orbiter Level 1b Data Set
FIT Image
GRIdded Binary (.grb)
Raster Matrix Format
EUMETSAT Archive native (.nat)
Idrisi Raster A.1
Intergraph Raster
Golden Software ASCII Grid (.grd)
Golden Software Binary Grid (.grd)
Golden Software 7 Binary Grid (.grd)
COSAR Annotated Binary Matrix (TerraSAR-X)
TerraSAR-X Product
DRDC COASP SAR Processor Raster
R Object Data Store
Portable Pixmap Format (netpbm)
USGS DOQ (Old Style)
USGS DOQ (New Style)
ENVI .hdr Labelled
ESRI .hdr Labelled
Generic Binary (.hdr Labelled)
PCI .aux Labelled
Vexcel MFF Raster
Vexcel MFF2 (HKV) Raster
Fuji BAS Scanner Image
GSC Geogrid
EOSAT FAST Format
VTP .bt (Binary Terrain) 1.3 Format
Erdas .LAN/.GIS
Convair PolGASP
Image Data and Analysis
NLAPS Data Format
Erdas Imagine Raw
DIPEx
FARSITE v.4 Landscape File (.lcp)
NOAA Vertical Datum .GTX
NADCON .los/.las Datum Grid Shift
NTv2 Datum Grid Shift
ACE2
Snow Data Assimilation System
Swedish Grid RIK (.rik)
USGS Optional ASCII DEM (and CDED)
GeoSoft Grid Exchange Format
Northwood Numeric Grid Format .grd/.tab
Northwood Classified Grid Format .grc/.tab
ARC Digitized Raster Graphics
Standard Raster Product (ASRP/USRP)
```

```
Magellan topo (.blx)
SAGA GIS Binary Grid (.sdat)
Kml Super Overlay
ASCII Gridded XYZ
HF2/HFZ heightfield raster
OziExplorer Image File
USGS LULC Composite Theme Grid
Arc/Info Export E00 GRID
ZMap Plus Grid
NOAA NGS Geoid Height Grids
```

Appendix G

Aggregation over raster files

```
# The programming language is Python
# The author of this program is Farzin Ashouri
from osgeo import gdal, gdalconst, gdal_array
from osgeo.gdalconst import *
import numpy, os, sys, time
import scipy.stats as stats
#  Driver names (code) are available at   http://www.gdal.org/formats_list.html
def aggImage(fn_a, fn_b):
    gdal.AllRegister()  # Register all drivers at once
    dsa=gdal.Open(fn_a, GA_ReadOnly)
    dsb=gdal.Open(fn_b, GA_ReadOnly)
    colsa=dsa.RasterXSize
    # No parentheses - because they're properties not methods
    rowsa=dsa.RasterYSize
    colsb=dsb.RasterXSize
    # No parentheses - because they're properties not methods
    rowsb=dsb.RasterYSize
    if dsa is None:
        print 'Could not open the image'
        sys.exit(1)
    if dsb is None:
        print 'Could not open the image'
        sys.exit(1)
    band1a = dsa.GetRasterBand(1)
    dataa = band1a.ReadAsArray(0, 0, colsa, rowsa).astype(numpy.float)
    type(dataa)
    band1b = dsb.GetRasterBand(1)
    datab = band1b.ReadAsArray(0, 0, colsb, rowsb).astype(numpy.float)
    data_a=gdal_array.LoadFile(fn_a)
    data_b=gdal_array.LoadFile(fn_b)
    data_a=data_a.astype(float)
    data_b=data_b.astype(float)
    data_a[data_a==0] = numpy.nan
    data_b[data_b==0] = numpy.nan
    data_a_row = data_a.reshape(1, rowsa*colsa)
    data_b_row = data_b.reshape(1, rowsa*colsa)
    C = numpy.array([data_a_row, data_b_row])
    C=stats.nanmean(C)
    C = C.reshape(rowsa, colsa)
```

```
C[numpy.isnan(C)] = 0
gdal_array.SaveArray(C, 'out.tif', 'GTiff', gdal.Open('a.tif'))
os.system('gdalwarp -dstnodata None out.tif out_none.tif')
```

Appendix H

Moving aggregated values function

```
CREATE OR REPLACE FUNCTION publicdata.climate_aggre_mov(envvar text,
 x double precision, y double precision, src_srs integer,
trg_srs integer, start_date timestamp without time zone,
 end_date timestamp without time zone, preced integer, follow integer)
  RETURNS SETOF record AS
$BODY$WITH
P as
( SELECT st_transform((st_GeomFromText
       ('POINT(' || $2::text || ' ' || $3::text || ')',$4)), $5) as point),

D as
( SELECT (publicdata.get_envvar_vals_for_pt($1, P.point)).doy,
         (publicdata.get_envvar_vals_for_pt($1, P.point)).val
  FROM P),

F as
( SELECT all_dates.date_col, min(D.val) as minval
  FROM ( SELECT date_trunc('day', dd):: date as date_col
         FROM generate_series($6::timestamp, $7::timestamp,
                              '1 day'::interval) dd) as all_dates
         LEFT JOIN D ON all_dates.date_col =D.doy
         GROUP BY date_col
         ORDER BY date_col )


SELECT date_col AS greatest_id_in_group, minval, avg_last_window_inclusive,
        stv_last_window_inclusive, cnt_last_window_inclusive
  FROM ( SELECT date_col, minval,
                avg(minval) OVER
                   (ORDER BY date_col ROWS BETWEEN $8 PRECEDING AND $9 FOLLOWING)
                               AS avg_last_window_inclusive,
                stddev(minval) OVER
                      (ORDER BY date_col ROWS  BETWEEN $8 PRECEDING AND $9 FOLLOWING)
                               AS stv_last_window_inclusive,
                count(minval) OVER
                     (ORDER BY date_col ROWS  BETWEEN $8 PRECEDING AND $9 FOLLOWING)
                               AS cnt_last_window_inclusive,
                (((to_char(date_col, 'J')::integer)-
                (to_char($7::date, 'J')::integer))) as m
```

```
        FROM F ) as x
        --WHERE m % ($8+$9+1) = 0

CREATE OR REPLACE FUNCTION publicdata.climate_aggre_Multiple_Years
                          (envvar text, x double precision, y double precision,
                          src_srs integer, trg_srs integer,
                          start_date timestamp without time zone,
                           end_date timestamp without time zone, agg_level integer)
  RETURNS SETOF record AS
$BODY$WITH
P as
( SELECT st_transform((st_GeomFromText
                     ('POINT(' || $2::text || ' ' || $3::text || ')',$4)), $5)
                      as point),

D as
( SELECT (publicdata.get_envvar_vals_for_pt($1, P.point)).doy,
         (publicdata.get_envvar_vals_for_pt($1, P.point)).val
  FROM P),

F as
( SELECT all_dates.date_col, min(D.val) as minval
  FROM ( SELECT date_trunc('day', dd):: date as date_col
         FROM generate_series($6::timestamp, $7::timestamp, '1 day'::interval) dd)
                           as all_dates
         LEFT JOIN D ON all_dates.date_col =D.doy
         GROUP BY date_col
         ORDER BY date_col ),

H as
(SELECT n, date_col AS greatest_id_in_group ,
        avg_last_window_inclusive, stv_last_window_inclusive
  FROM ( SELECT date_col,
                avg(minval) OVER (ORDER BY date_col ROWS ($8-1) PRECEDING)
                                  AS avg_last_window_inclusive,
                stddev(minval) OVER (ORDER BY date_col ROWS ($8-1) PRECEDING)
                                  AS stv_last_window_inclusive,
                (extract(doy from date_col)::integer) AS n,
                (((to_char($7::date, 'J')::integer)-
                (to_char(date_col, 'J')::integer))) as m
         FROM F ) as x
  WHERE n % $8 = 0 )
SELECT  n, greatest_id_in_group, avg_last_window_inclusive, stv_last_window_inclusive
FROM    H
WHERE   n = (extract(doy from (SELECT max(greatest_id_in_group)
                           FROM H)::date)::integer)

$BODY$
```