Back End Support for Trajectory Management Using Open Source Software

Lizda Iswari

March, 2010

Back End Support for Trajectory Management Using Open Source Software

by

Lizda Iswari

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

Degree Assessment Board

Thesis advisor	Dr. U. D. Turdukulov Dr. J. Morales
Thesis examiners	Dr. Ir. R.A. de By Dr. N. Meratnia

INTERNATIONAL INSTITUTE FOR GEO-INFORMATION SCIENCE AND EARTH OBSERVATION

ENSCHEDE, THE NETHERLANDS

Disclaimer

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

Abstract

Trajectory is defined as a movement of an object in space and time. It consists of a set of spatial and temporal data. Exploration of this data set can be used to express the movement behaviour of moving objects that is previously unknown. Trajectory data exploration can be conducted based on their geometric and semantic properties. The geometric properties related to the characteristics of the trajectory are travelled time, travelled distance, average speed and direction of movement. The sematic properties can be obtained from the requirements and needs of users of the trajectory data.

This research proposes several alternatives on how to manage and explore trajectory data in a database system. Research has been conducted in two stages. The first stage deals with designing the data models. There are two models involved, a general trajectory model and an application-specific data model. These data models are designed as UML class diagrams that specify the required attributes and methods to manage trajectories in a database system. The second stage is associated with the provision of back end support to extract the geometric and the semantic properties of trajectory. The iceberg movements in Antarctica have been chosen as a case study. Based on the analysis of user requirements, there are some necessities deal with: managing data errors in the data set, extracting geometric properties of the iceberg, detecting iceberg events and analyzing behaviour movement of icebergs.

To meet these requirements, four type of database functions were implemented in the PostgreSQL database system. The first functions deal with data pre-processing, i.e. to clean data set from inconsistencies, empty values, duplicates and outliers. The second functions deal with extracting the characteristic of an iceberg. The third functions deal with classification of event during the lifespan of an iceberg and functions to detect the calving occurrences. The last functions deal with trajectory data mining to find similar patterns of iceberg movement based on some criteria that are determined by the users. All these functions still consider trajectory as an individual entity. Further research is needed to provide back end support that also involve neighbourhood information and relationship to other trajectories that enable data exploration based on spatial and temporal proximity.

Keywords

back end support, iceberg event, data pre-processing, similar pattern, trajectory data mining

Acknowledgements

Alhamdulillahi Rabbil 'Alamin. Praise to Allah subhanahu wa ta'ala, the Most Merciful and Gracious, to Whom I send my prayers and hope.

Firstly, I would like to express my gratitude to my first supervisor Dr. Ulanbek Turdukulov. Thank you for the great guidance, discussion and always welcome anytime I have questions and difficulties. I also would like to extend my gratitude for my second supervisor Dr. Javier Morales who has helped me in technical part of my thesis. From them I learned many about the research. Thank you for the invaluable remarks, suggestions and comments that you have given to me.

Secondly, I would like to acknowledge to the Ministry of Education Republic of Indonesia who have sponsored my study in the Netherlands. I also would like to send my appreciation for Dirgahayu family. They have assisted me in many ways. Thank you for always encourage and open my understanding on algorithms. May Allah bless your family.

Thirdly, to all my GFM2 colleagues thank you for the nice friendship that we have shared and also for sharing the knowledge among us. For my all Indonesian friends, thank you for making me feel still in home. All of us had experienced many hard and happy times. I am sure we will miss the moment that we had spent in the Netherlands.

Moreover, for the whole staff of ITC. Thank you for the knowledge, guidance and help that has given during my stay in the Netherlands.

Finally, my deepest gratitude for my mother Hajjah Masitah and my husband Agung Nugroho Adi. Thank you always send me prayers and always support in any condition that I have. Also to all my family in Banjarmasin and Yogyakarta. May Allah always bless our lives.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Problem Statement

Development of communication and positioning technology has provided many advantages in scientific research. One of the associated benefits is the ability to record and track the movement of objects in space and time called the trajectory [GS05, WXCJ98]. Trajectory consists of a large number of spatial and temporal data. Exploration of this data set can be used to express the movement behaviour of moving objects that is previously unknown. Spaccapietra at [SPD+08] states that the trajectory analysis is the key to applications aimed at a common understanding and management of the complex phenomena that involve moving objects. Some examples of trajectory analysis can be found on the hurricane trajectories to study climate change and the wildlife trajectories to study the causes of their migration.

In order to explore the knowledge of the trajectories, these data sets need to be managed in a database system. However, representing and querying trajectories in a database system, which have to be matched with user needs and requirements, is not an easy task. There are some challenges in managing trajectories, such as how to manage and update dynamic and large amount of trajectory data, how to maintain trajectories consistency from potential error of missing value and data duplicate, how to mine trajectories for knowledge discovery, and how to query the continuously changing spatial and temporal data of trajectories [GS05, MS04]. Although the existing database systems have provided some functionality to represent spatial and non spatial data and also to explore both data for discovering pattern and unknown knowledge, until recent days, these functionalities can not be applied instantly for trajectories' phenomena that has dynamic change [SPD+08, BPT04].

Iceberg movement in Antarctica is an example of trajectory that needs to be managed in a database system. The iceberg movements have been recorded since 1978 until present by the National Ice Center [Cen]. A broad study of distribution and behaviour of icebergs has been conducted which is stimulated by natural phenomena like global warming and climate change, environmental problems, navigation purposes, and engineering activities that are endangered by floating or grounded icebergs [BTKP09]. Problems exist when users want to analyze and use iceberg data with regard to their inconsistency and irregularity.

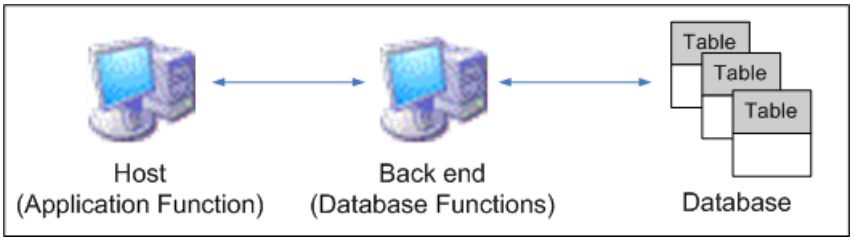

Figure 1.1: Back end support as database functions that connect the application to the required resource (modified from: (Mar80))

Some problems are missing values that lead to data gaps, typing error that create outliers, a naming system that triggers difficulty to trace the ancestor of iceberg before it brakes into pieces, and irregularity of temporal resolution that causes complexity of data integration. Although some studies have been proposed in investigating the iceberg phenomena [BL02, BWM07], none of them deal with managing large amount of dynamically changing iceberg data and providing functionality, such as checking their consistency, providing queries for tracing movement of iceberg and discovering their behavioural patterns.

This research proposes some alternatives of how to deal with trajectory's problems mentioned above. It is conducted by providing some functions that are acted as back end support to manage trajectories in a database system. The term of back end support can be described literally as database functions. It has a role as data processor which interacts directly with the data source. User can only interact with back end through front end application or high level manipulation language, i.e. SQL syntax. Maryanski in [Mar80] have illustrated the basic form of back end support in a database system that consist of a locally connected pair of computers. As can be seen on figure 1.1, the application programs are executed by the host computer. When these programs request some data from a database, they have to contact a server that holds database functions controlling access to a database.

Regarding trajectory management, some back end support is needed for data pre-processing and data extraction. Data pre-processing deals with managing data at the initial stage of data storage, update and consistency check. The data extraction support deals with information that can be derived from trajectory, such as its lifespan, average speed, and moving direction. These main supporting functions are mainly focused on management of trajectories from geometric perspective [ABdM$^+$07].

The further needs on trajectory also address some necessities of the semantic properties which can be obtained through analysis of user requirements. Techniques from data mining, such as clustering, classification, aggregation and pattern recognition can be employed to reveal the required semantic properties. It may also possible that further exploration on the semantic properties can derive to the new knowledge of trajectory.

Implementation of back end support of trajectory management can not be separated from database management systems (DBMS) that have capabilities to handle spatial and temporal data. Although there are DBMS that have al-

ready provided some spatial and temporal libraries (built-in functions), selection of DBMS to develop these needed supports should also be considered. The first priority is put on open source software. Nowadays, open source software have been recognized and used among software developers and end-users. It has a community that contributes the software development. As an example is PostgreSQL that has been developed by a collaborative work of public users. Current version of PostgreSQL has provided some features to handle spatial/-geometry data type through a package termed as PostGIS [Pro09]. In addition, it also features a simple way to represent spatial data, has a compliance with OGC (Open Geospatial Consortium) spatial data standards and have the possibility to integrate with other platforms (interoperability) [Gro09]. By implementing trajectory back end support using the open source DBMS, it will have the features mentioned above in further development.

1.2 Research Identification

The aim of this research is to provide back-end supports for managing trajectories using open source database management system.

1.2.1 Research Objectives

1. To provide conceptual data model of trajectories.
2. To extent database functionality to operate on trajectories.
3. To apply back end support on a case study.

1.2.2 Research Questions

1. To provide conceptual data model of trajectories.
   (a) What data types, attributes, operations, classes, and relationship are needed to represent general trajectories?
2. To extent database functionality to operate on trajectories.
   (a) What functions are needed for extracting geometric properties from trajectories?
   (b) What functions are needed for extracting semantic properties from trajectories?
3. To apply back end support on a case study.
   (a) What are the user requirements of icebergs trajectory?
   (b) What is the required data model to represent icebergs trajectory?
   (c) What steps are required for pre-processing data?

(d) What functions are used to detect iceberg events, such as calving, grounded and floating; and also parent child relationship among icebergs?

(e) What functions are used to detect similar movement patterns of iceberg?

1.2.3 Innovation Aimed At

The novelty of the research is in designing a data model of trajectory and implement the model as database functions using open source database management system (DBMS). These functions can be applied for extracting the geometric and semantic properties of icebergs trajectory in Antarctica.

1.2.4 Related Work

Several research efforts have been carried out on management of trajectories. Some include modelling and representing trajectories [WXCJ98, GS05, SPD+08] and mining pattern of trajectories [BPT04, LI02, LIW05]

Güting in [GS05] described that moving object as part of trajectory can be modelled based on two perspectives: location management and in a spatio-temporal database. The first perspective focuses on time-dependent locations [WXCJ98]. Data was developed as dynamic attributes within a data model which is called MOST (Moving Object Spatial Temporal) and can be queried through a query language called FTL (Future Temporal Logic). This model was restricted to one spatial data type, moving object point, and the query language can only be used to trace current and near future movement [GS05]. The second perspective focuses on time-dependent geometries [GBE+00, GS05]. This model was proposed under the CHOROCHRONOS project [GBE+00, GS05]. The team project developed a first model by putting moving line and moving region as additional spatial data types. They also provided additional query language primitives to trace the historic movement of object. Both of these two perspectives developed as moving object database (MOD).

While Güting and Wolfson focused on modelling moving object based on geometry properties, Spaccapietra in [SPD+08] have proposed conceptual modelling that combine geometry with semantic properties of trajectory. Trajectory is modelled or structured into countable semantic units. There are two proposed modelling approaches, design pattern and dedicated data types. Trajectory design pattern includes all object types for representing trajectory. Each application may have different model. Spaccapietra described design pattern as a half-baked schema with respect to pattern modification to make it fully compliant to application requirements. The next model is trajectory data types which try to encapsulate trajectory data into a dedicated *TrajectoryType* data type and provide methods to access trajectory components. This dedicated data type only handle the geometric properties of the trajectory, the semantic properties that complements trajectories for a given application is described as attributes and relationships of objects that are involve in application.

To extract knowledge from trajectories can be done through trajectory data mining. Laube in [LI02, LIW05, LvKI05] introduced relative motion (REMO) patterns that can be found in groups of moving point objects. The concept of REMO analysis is done by comparing the motion parameters, i.e. motion azimuth, speed, and change of speed; of different objects at different times. A pattern is defined as a search template that can span over time, across objects, or combination of both search templates.

Despite of general concept of how to manage trajectories mentioned above, the chosen of trajectory representation has to consider the nature of the data set. Not all proposed concepts are compatible when they are applied to specific trajectories. Iceberg, in this case, has different characteristic compare to common moving entities, such as car or animal. Iceberg movements does not have to be in a constrained spatial network. Furthermore, record of iceberg movements may also be done in a larger temporal resolution, such as in days or weeks. Therefore, these characteristics have to be considered when designing data structure of icebergs trajectory and implementing its needed back end support.

1.3 Method Adopted

The steps to conduct the research can be seen on figure 1.3. Briefly, each steps can be explained as follow:

1. Literature review.

   The main literatures that will be used in this research are from fields of moving object database, trajectory data modeling, trajectory data mining and similarity search in trajectories.

2. Analysis of user requirement.

   Analysis of user requirement on iceberg movements. There are four types of requirements, i.e. data pre-processing on iceberg data set, trajectory data extraction, iceberg events detection, and analysis of similar pattern on iceberg movement.

3. Design conceptual data model.

   Based on analysis of user requirements, conceptual data model are designed into two types: general and application specific data models. These models are represented as trajectory design pattern and UML class diagram.

4. Implementation of back end support for data pre-processing.

   There are some database functions can be used to identify and manage data inconsistencies, empty values, data duplicates and outliers.

5. Implementation of back end support for trajectory data extraction.

   There are some database functions are needed to extract general information from trajectory, such as travelled time, travelled distance, speed and movement direction.

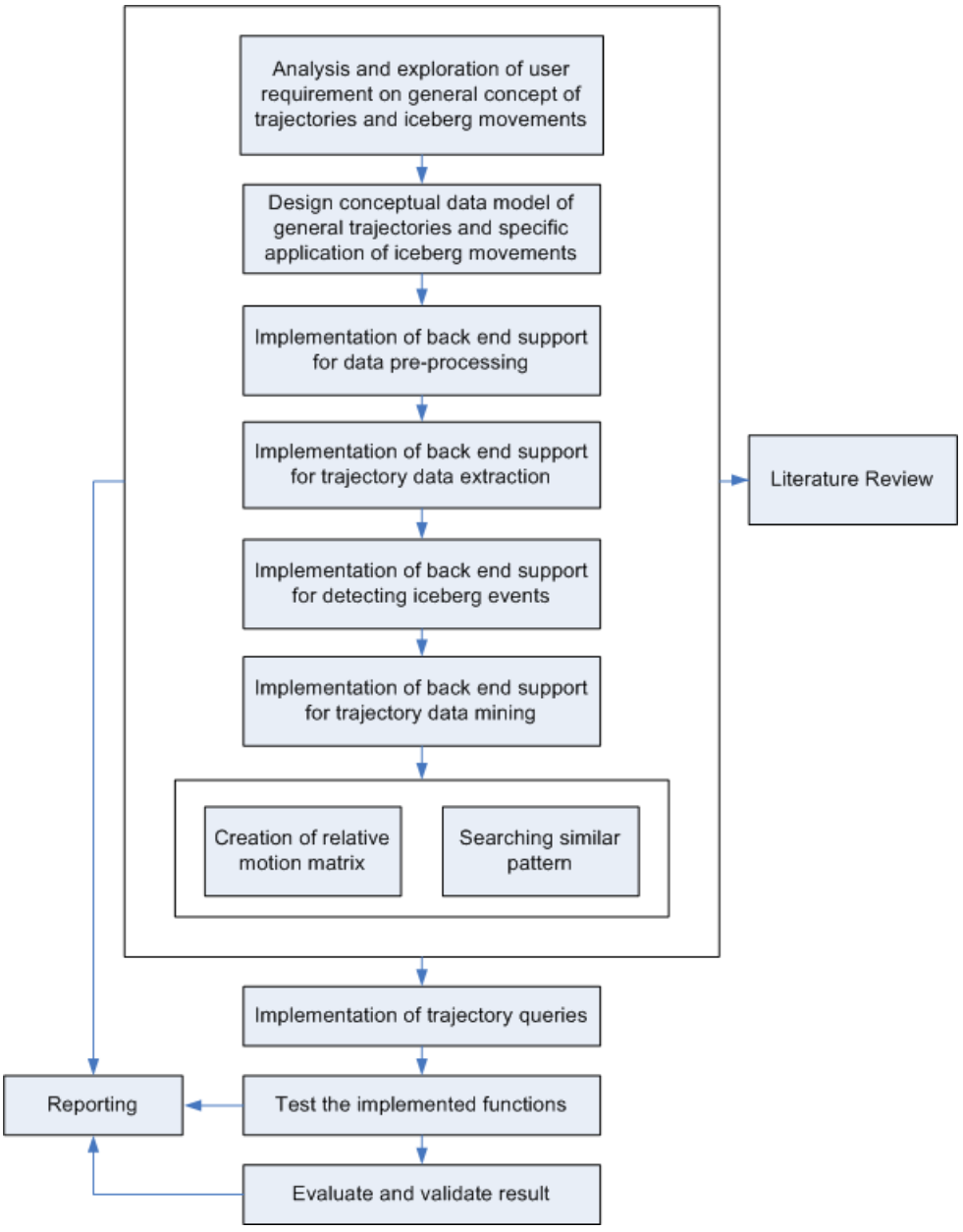

Figure 1.2: Steps for conducting research

6. Implementation of back end support for iceberg event detection.

   This back end support is related with specific requirement from icebergs trajectory to detect events that may occur to an iceberg. These events are classified as calving, grounded and floating. From calving detection, parent child relationship among icebergs may also be identified.

7. Implementation of back end support for trajectory data mining.

   This back end support is dealing with database functions for discovering similar patterns of iceberg movements. There are three type of patterns that are applied with respect to the concept of relative motion pattern [LI02, LIW05]. These patterns are detected over time, across objects and combination of these two parameters, i.e. over time and across object.

8. Test the implemented database functions.

   Providing some queries that can be used to extract general information of trajectory, to detect iceberg events and to search similar pattern of iceberg movement.

9. Evaluate and validate the results.

   System evaluation and validation is conducted by exploring and manipulating icebergs data set, extracting information based on the needs of user requirements and running queries on it.

1.4 Structure of Thesis

Based on the method adopted, research is organized in five chapters as follows:

Chapter 1 describes the motivation to do this research, states the problem, research objectives and questions and also the method adopted to conduct the research.

Chapter 2 discusses user requirements toward iceberg trajectory and literature background of how to represent and mine trajectory data. User requirements are divided into four groups, i.e. data pre-processing, trajectory data extraction, iceberg event detection and trajectory data mining. This chapter also describes some approaches to represent trajectory based on its geometric and semantic properties.

Chapter 3 contains trajectory data modelling which adopts the concept of trajectory design pattern. This chapter also discusses representation of trajectory by using UML class diagram.

Chapter 4 discusses implementation of trajectory back end support. In this chapter, there are list of functions that have been implemented to check and manage data errors, to extract trajectory data, to detect iceberg event and to discover similar pattern of iceberg movements.

Chapter 5 presents the conclusions that are drawn after execution of database functions on iceberg trajectory and the recommendations which are made for future improvement of trajectory back end support.

Chapter 2

Application Requirements and Concept of Trajectory

This chapter highlights the case study and the literature background of trajectory. In the first section (2.1) user requirements on iceberg trajectory are discussed. These user requirements are presented in the list of questions and are also illustrated in UML use case diagram. The second section (2.2) describes some concepts on trajectory. Trajectory is described as an entity that has geometric and semantic properties. The geometric properties are general information that can be derived directly from spatial and temporal attributes of trajectory, such as moving direction and average speed. While for revealing the semantic properties, trajectory data need to be explored through trajectory data mining which is discussed in section 2.3 with provision of technique to discover similar patterns in trajectories.

2.1 The Antarctic Iceberg

Icebergs are a result of big masses of ice separating from either a glacier or an ice shelf. They come in many shapes, such as dome, pinnacle, and blocky; and sizes, from ice-cube size to ice islands that are the size of a of a small country. The term iceberg refers to a piece of ice that has diagonal of size larger than 5 meters. Typically, icebergs are found in open seas and formed during spring and summer when warmer weather increases the rate of iceberg calving from its ice shelf. From the whole part of an iceberg, only 1/8th is above the waterline and the rest is under the water surface. The upper part consists of snow and it is not very compact. Meanwhile the bottom part is located in the cold core which makes it very compact and relatively heavy [SC09].

Icebergs are found mostly in Antarctica. Almost ninety percent of the world's mass of iceberg is found surrounding the Antarctic and ninety-eight percent of the Antarctic continent is covered by ice [Bri]. Antarctica is the earth's southernmost continent located on the South Pole. The Antarctic icebergs calve from floating ice shelves. Floating ice shelves are a continuation of the flowing mass of ice that makes up the continental ice sheet.

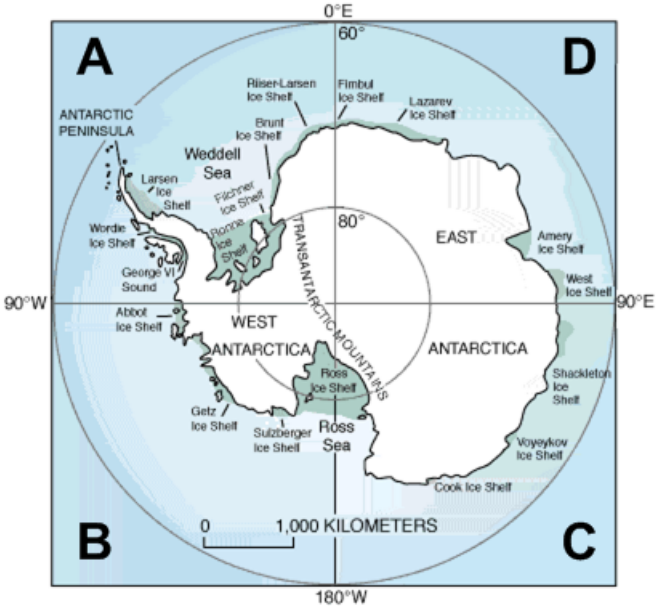

Figure 2.1: The Antarctic quadrant as the basis for naming the iceberg (taken from: www.solcomhouse.com)

2.1.1 Iceberg Data Set

Icebergs in Antarctica are named according to the Antarctic quadrant in which they were originally sighted (see figure 2.1). The quadrants are divided counterclockwise in the following manner [Cen]:

- A = 0° - 90°W (Bellinghausen/Weddell Sea)
- B = 90°W - 180° (Amundsen/Eastern Ross Sea)
- C = 180° - 90°E (Western Ross Sea/Wilkesland)
- D = 90°E - 0° (Amery/Eastern Weddell Sea)

There are two main organizations that record icebergs existence in Antarctica, i.e. the U.S. National Ice Center (NIC) and the Brigham Young University (BYU). They use a variety of remote sensing images and sensors to track iceberg positions and other characteristics. However, only NIC has the authority to coordinate iceberg positions and track all icebergs based on three basic requirements [BL02]:

1. Iceberg size has to be at least 10 nm (nautical miles), equal to 18.5 kilometers, along the major axis.
2. The most recent sighting of iceberg must have occurred within the last 30 calendar days.
3. Iceberg location has to be positioned south of 60° south latitude.

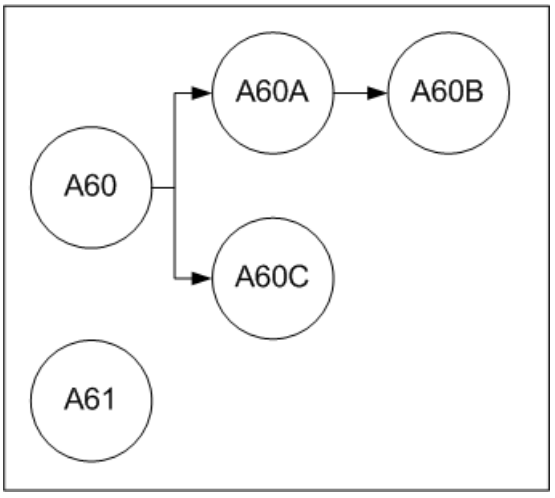

Figure 2.2: The approach of how NIC defines the name for icebergs in Antarctica

NIC will remove record of iceberg from database if the iceberg has size reduction in which it becomes less than 10 nm and loss of visual sighting for 30 consecutive days if it is positioned on the north of 60° south latitude.

Furthermore, NIC also has the authority to name the icebergs. When an iceberg is first sighted and meets the requirements above, NIC documents this iceberg based on its quadrant name along with a sequential number. However, if the new found iceberg is a result of calving (from an already identified iceberg), this iceberg will be named by adding an alpha suffix to the name of its "parent". Examples of how NIC names the iceberg based on these rules can be found in the following illustration (assume the last tracked iceberg in quadrant A is A60):

1. If there is a new found iceberg located between 0° - 90°W (Quadrant A) and meet NIC criteria mentioned above, it will be named as A61.

2. If the new found iceberg has calved from A60, it will be named as A60A. In this case, A60 is defined as the parent of A60A.

3. If A60 calves after A60A already calved into A60B, the new found iceberg is named as A60C.

The names of A60A, A60B and A60C mark the calving events of icebergs which belong to the group of A60. These "new" icebergs include the parent iceberg's original name and an alpha suffix in alphabetical order. Figure 2.2 illustrates how NIC names the calving icebergs.

Record of icebergs movements has been provided since 1978. This data set is downloadable for free from NIC website [Cen]. There is some basic information, such as iceberg name, iceberg position in latitude and longitude, time of recording in the format of long integer (four digits of the year and three digits of the day sequence in that year), iceberg size in nautical miles, and satellite name that is used to record icebergs movements. The temporal resolution is

delivered in irregular time steps. For two consecutive records of an iceberg, the difference may be 5 to 30 days. However, there are found some records which have more than 30 days as its temporal resolution.

The Antarctic iceberg data set consist of 301 icebergs with 15737 records. They spread over four quadrants (see figure 2.1). Initial inspection on this data set shows there are data errors that are caused by data inconsistencies, duplicates, typing errors and missing values.

Some approaches have been addressed to manage these errors. Blok in [BTKP09] described that to manage the missing value in longitude or latitude had been done by simple interpolation based on neighbouring space-time positions. Duplicates have been deleted manually by visual inspection of the trajectory plotted in the browser, which may be due to duplication of typing errors. Visual inspection also revealed some out of context positions along the trajectory of some icebergs. They are assumed as records that have incidental exchange of plus or minus signs of latitude and/or longitude. Some of these clear errors could be corrected by considering the location of its previous and next records.

The approach mentioned above, however, has a lot of limitation. The first limitation is on the number of records that may be employed in the prototype. This is related with the prototype performance that gets slow and even unresponsive when all records are entered. The second limitation is related to the inability of the prototype support software to adopt the nature of the data set. E.g. the Antarctic iceberg has its own projection system which can not be accommodated directly by the visualization software, such as Google Earth. The last limitation refers to dynamic updating and data storage. Blok described that on the proposed prototype, data input is still managed in flat files which create difficulty for data updating and storage capacity. Hence, management under database technology is recommended as an alternative to handle errors before the users can further explore the data set.

2.1.2 User Requirements toward Iceberg Data Set

The iceberg data set from NIC has been used by researchers with various objectives. As Ballantyne mentioned in [BL02], this data set has been used for understanding the trend of climate change, for analysing several major calving events and for navigation purposes around Antarctica. In this section, user requirements toward iceberg data set are discussed. There are four main requirements that are dealing with data pre-processing, requirement for extracting general information of trajectory, requirements on analyzing event detection and requirements that are analyzing similarity pattern of iceberg movements.

User Requirements on Data Pre-processing

Iceberg records that are prepared by NIC are raw data that still need to be refined before they can be used for further purposes. Data pre-processing is required to provide a good quality and a consistent data set. It consists of some sub-processes [HK06]:

1. Data cleaning: clean data set from errors, such as inconsistencies, empty values, typing errors and outliers.

2. Data integration: integration of multiple databases or files.

3. Data transformation: data normalization and aggregation.

4. Data reduction: obtains reduced representation in volume but produces the same or similar analytical results.

In the following is the list of preliminary questions that users may ask toward iceberg data set before they go on further data process:

1. How to identify and manage duplicates?

2. How to fill in the missing values?

3. How to identify and remove the outliers?

4. How to identify and manage the typing errors?

5. Do all attributes ready to be used for data computation?

6. Do all attributes have consistent data types?

User Requirements on Trajectory Data Extraction

Iceberg data set identifies that there is dynamic movement behaviour of iceberg. The movement of iceberg can be observed when an iceberg has changed its position from the last time it was sighted. This is termed as drifting iceberg. Some early observations and measurements of drifting icebergs were made in support of the development of practical forecasting tools [VD00]. When an iceberg drifts, it also creates a trajectory as the path of its movement. Some questions may be asked on iceberg trajectory, such as:

1. How far an iceberg has drifted?

2. How long an iceberg has drifted?

3. What is the average speed of the drifted iceberg?

4. To which direction an iceberg has drifted?

5. What is the complete trajectory of an iceberg during its lifespan?

6. How many records of iceberg movements have been documented during its lifespan?

User Requirements on Event Detection

Schmittner in [SYW02] describes iceberg information as invaluable factors for climate related investigations. Climatologists analyze the indicator of climate change based on iceberg numbers and events that are occurred to icebergs, such as calving, appearance, disappearance, grounded, or moving. Many scientists try to investigate these events by defining the model of iceberg motion with respect to external factor and the life history of icebergs. There are a number of question regarding to iceberg events, such as:

1. When did the icebergs calve from its parent?
2. Where did the calving site?
3. How many icebergs that have calved?
4. Which iceberg is grounded?
5. Are the number of icebergs increased?

User Requirements on Trajectory Data Mining

Pelekis in [PKM+07] described that trajectory similarity search forms an important class of queries in trajectory data analysis and spatio-temporal knowledge discovery. Regarding to iceberg trajectory, this analysis is required to describe patterns in iceberg behaviour. Many users, such as navigator, climatologist, and glaciologist may use this analysis which is an essential information for avoiding any destruction consequences, such as to define a new safe shipping lanes in navigation area. Some basic questions of behaviour patterns are:

1. How to define similar moving pattern from iceberg data set?
2. When did an iceberg have a similar pattern?
3. Are there any clusters of iceberg that follow similar patterns of movement at similar times?

For answering the questions above, data need to be explored and analyzed. These activities are strongly related with data mining which offers some techniques to discover patterns of movement behaviour (see section 2.3).

Use Case Diagram of Iceberg Trajectory

User requirements that have been discussed above can be summarized into UML (Unified Modelling Language) use case diagram. As it can be seen on figure 2.3, this diagram involves three types of actors: NIC as data provider, iceberg specialist and climatologist that has role as data analyst as well as data user and navigator as data user.

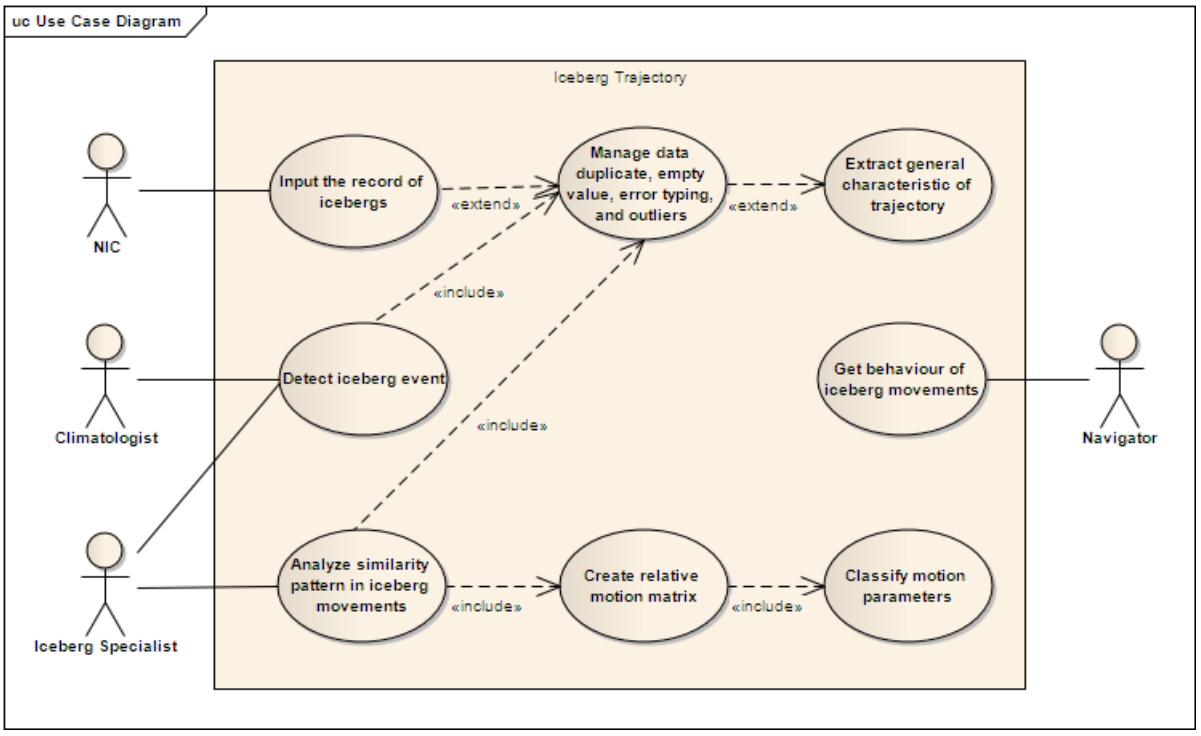

Figure 2.3: The user requirements toward iceberg data set

- NIC has responsibility to input the record of iceberg movements. This input is considered as a raw data that needs to be refined in data pre-processing before it can be extracted into meaningful information of trajectory.
- Climatologist and iceberg specialist have necessity to detect the iceberg events. One of data source is iceberg data set, therefore action for detecting iceberg event also includes action of data pre-processing and data extraction.
- Iceberg specialist also has necessity on analyzing the behaviour of iceberg movement. This analysis includes data mining technique, i.e. classification of motion parameters, and put these parameters into semi-spatial temporal matrix to search the similar pattern of movement.
- Navigator has necessity to define the new shipping lanes based on analysis of behaviour of iceberg movements.

2.2 Concept of Trajectories

Trajectory is defined as the path made by object or moving entity through the space where it moves [AAPS08]. If t_0 is defined as the moment of path started

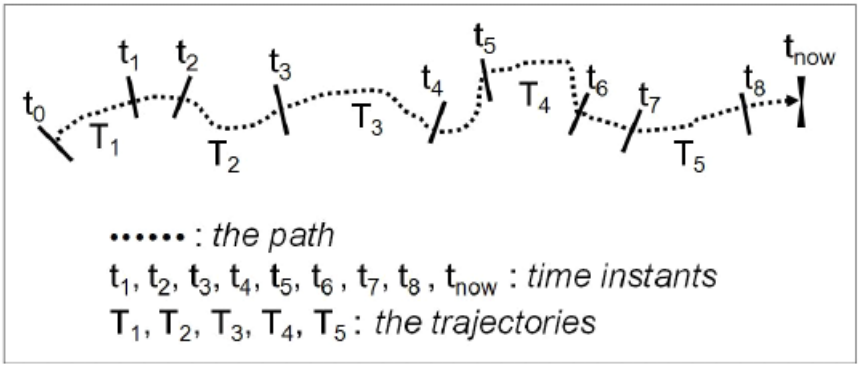

Figure 2.4: Trajectory definition as spatio-temporal path which consist of a sequence of moves and stops (taken from: (SPD+08))

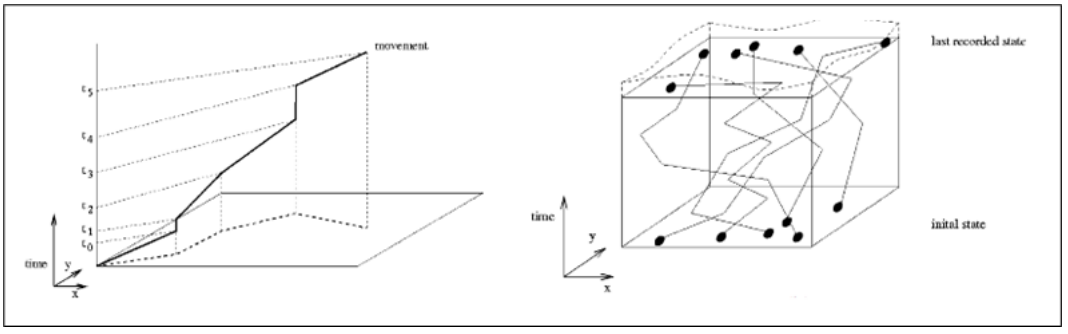

Figure 2.5: Trajectory definition as time space function that has movement along temporal axis in upward direction (taken from: (BPT04))

and t_{end} is defined as the moment when it ended, for any moment t_i between t_0 and t_{end}, there is a position in space that was occupied by the object at this moment. During the lifespan of trajectory, trajectory may semantically be segmented by defining a temporal sequence of time sub-intervals where object position changes (moves) and stays fixed (stops) [SPD+08]. Hence, an object may travel to a number of trajectories as a sequence of moves going from one stop to the next one (see figure 2.4).

Trajectories may also be seen as a function that matches time moments with position in space. By using a space-time-cube [BPT04], trajectory can be described as the path that moves along the temporal axis in upward direction (see figure 2.5).

Figure 2.5 shows a single trajectory on the left side and several trajectories evolving in a finite region on the right side. Both of these trajectories may consider as time space function. The top of the cube is the time of the most recent position sample and the wavy-dotted lines represent the growth of the cube with time.

Spaccapietra in [SPD+08] defined there are two facets of a trajectory:

- A geometric facet : the spatio-temporal recording of the position of the

traveling point (this research only consider the point geometry).
This facet can be defined as a single continuous subset of the spatio-temporal path covered by the object's position during the whole lifespan of the object. It can also be represented as a time space function which is a continuous function from a given time interval into a geographical space (the range of the function). Time space function is described as:

trajectory : $[t_0, t_{end}] \rightarrow$ *space*

- A semantic facet : the information that conveys the application oriented meaning of the trajectory and its related characteristics.
  The concern of this facet is to give a meaning or semantic interpretation of the application object. There are three components of trajectory that need to be defined in specific term based on application requirements. These components are stops, moves, and begin and end of the trajectory.

  - A stop is part of trajectory which is considered that the object has not effectively moved.
  - A move is part of trajectory that has a time interval and is delimited by two consecutive stops (s_1,s_2) where consecutive stops by definition must have non-overlapping time intervals.
  - Begin and End (t_0,t_{end}) are time interval of trajectory that is necessarily included in the lifespan of its traveling object and is necessarily disjoint from the time intervals of the other trajectories of the same object. Within (t_0,t_{end}) there is no instant that belong neither to a move nor to a stop.

2.2.1 Characteristic of Trajectory

Along the lifespan [t_0,t_{end}] or during the subset [t_i,t_{i+1}] path of object movement, trajectories can be analyzed to have a number of characteristics. These characteristics are defined by different properties depending on application requirements. The common characteristics are described as follow [AAPS08]:

1. Geometric shape of the trajectory in space
2. Traveled distance, i.e. the length of trajectory in space
3. Duration of the trajectory in time
4. Movement vector or major direction, i.e. from the initial to the final position
5. Mean, median and maximal speed
6. Dynamics (behavior) of the speed
   - Periods of constant speed, acceleration, deceleration, and stillness
   - Characteristics of these periods: start and end times, duration, initial and final positions, initial and final speeds.

- Arrangement (order) of these periods in time

7. Dynamics (behavior) of the directions
   - Periods of straight, curvilinear, circular movement
   - Characteristics of these periods: start and end times, duration, initial and final positions and directions, major directions, angles and radii of the curve.
   - Arrangement (order) of these periods and turning points in time.

These characteristics are important components to represent conceptual data model or schema of the application. Conceptual data modelling is an activity that focuses on reflecting the application requirements in a database design. The design of conceptual data model have to be complete and understandable for the user, so it can be translated into the logical data model, on the next step, without any further user input [TJ99]. For trajectories, the goal of developing a conceptual model is to provide constructs and rules that enable database designer to think about data in the model as a set identifiable trajectories travelled by application objects [SPD+08]. Hence, when designing the conceptual data model for trajectories, the employed characteristics and operations have to be fit with the application requirements.

2.2.2 Design of Trajectory

For designing conceptual data model of trajectories, there are two approaches that can be used [SPD+08]:

- Trajectory design pattern

  In this approach, trajectories and their components (stops, moves, begin and end) are represented explicitly in the database schema. This model supports database designer by providing a design pattern, i.e. a predefined sub schema that provides the basic data structure for trajectory modelling. By using this approach, database schema is easy to be understood and read by user since it looks like a normal spatio-temporal database. The database designer will not find difficulty to add the semantic information specific to the application and also to modify the initial schema for future needs. However, this approach requires implementation of many functions to access trajectory and its components, such as general attributes from trajectories and specific attributes from application requirements. These functions have to be coded by the application developers or by the users, which are required to write more complex queries, to access data from trajectories.

- Trajectory data type

  The second approach encapsulates trajectory data into a dedicated Trajectory data type. This data type is equipped with methods providing access to trajectory components (stops, moves, begin and end). This approach can be used to handles the geometric facet of the trajectory and the definition

of its components. However, it can not be used to manage semantic information since semantic information is application specific which can not be encapsulated into the data type. Hence, database designer has to define this information explicitly in the data model. In this approach, there are two kinds of trajectory data types that are proposed:

1. Data type TrajectoryType: an abstract data type that holds a number of properties and methods to describe a trajectory. The properties are consisting of: a time-varying point, a list of sample points, a list of stops, and a list moves. The methods are consisting of data acquisition mechanism, such as defineSamplePoints and defineStops, and methods for extracting additional information from trajectories, such as temporal extent and duration.
2. Data type TrajectoryListType : an abstract data type that is formed as a list of trajectories. This data type consists of a list of elements from TrajectoryType data type above with one requirement of temporal domain has to be disjoint or adjacent. The methods are adopted from generic List data types that are reformulated for trajectories, e.g. trajectoryAtInstant(t Instant) returns the trajectory that exist at instant t.

This research adopts trajectory design pattern for representing iceberg trajectories. This is due to various user requirements toward iceberg data set (see section 2.1.2. By using this approach, many use cases specific to iceberg data set can be represented in the conceptual data model. System developer or user may expand this model to fit with their needs. Furthermore, the geometric and semantic facets of iceberg trajectories may also be included in one compact design of iceberg trajectory data model.

Data modelling and processing methods that have been mentioned above deal with the geometric properties. They can be formed in general term since any trajectories can be applied to these concepts. Meanwhile, modelling and processing semantic properties of trajectories has not been addressed yet due to its application-dependent, whereas in fact, it may give meaningful pattern that can be extracted from trajectories. Extraction of meaningful pattern can be done through data mining. Techniques that are provided in data mining – predictive modelling, clustering, classification and association analysis – enable the process of discovering useful information from large data repositories such as trajectories. In the following section, data mining will be discussed with regard to discovery meaningful pattern or knowledge from trajectories.

2.3 Trajectory Data Mining

The massive amount of data in trajectories has possibility to be analyzed through discovery of patterns, i.e. patterns that show collective movement behaviour. Patterns will enable users to have a better understanding of the dataset or the original system by revealing some of its motion laws which are hidden in the chaos of trajectory representation [**?**]. The approach to turn trajectories into

patterns, more broadly as usable knowledge, is termed as trajectory data mining which is one part of knowledge discovery in databases.

2.3.1 Knowledge Discovery in Databases

Knowledge discovery in databases (KDD) is a combination of many research fields such as machine learning, pattern recognition, databases, statistics, artificial intelligence, expert system, data visualization, and high-performance computing [LIW05]. The universal goal of KDD is extracting high-level knowledge from low-level data in the context of large data sets [FPSS96]. KDD comprises many steps, which involve data preparation, search for patterns, knowledge evaluation, and refinement. These steps are repeated in multiple iterations as can be seen on figure 2.6 The central belief of KDD is that information is hidden in very large databases in the form of interesting patterns [MH09].

The process of KDD on figure 2.6 can be described as follow [FPSS96]:

1. Develop an understanding of the application domain and identify the goal of KDD based on user requirements.
2. Create a target data set by selecting a data set or data samples on which discovery is to be performed.
3. Data cleaning and processing, such as for removing duplicates and handling errors and missing values.
4. Data reduction and projection by finding useful features to represent data based on the goal of the task.
5. Matching the goals of the KDD process to a particular data methods, such as classification, regression, clustering, summarization, and so forth.
6. Choosing data mining algorithm(s) and selecting method(s) to be used for searching data patterns.
7. Searching for patterns of interest in a particular representation form, such classification rules, trees, regression, and clustering.
8. Interpreting mined patterns.
9. Using the knowledge directly or incorporate the knowledge into another system.

As can be seen on figure 2.6, KDD refers to overall process of discovering useful knowledge from data, and data mining refers to a particular step in this process. However, data mining is the central component that applies data analysis and discovery algorithms for extracting pattern from data. Giannotti in [**?**] describes the process in KDD, basically can be categorized into three main transformation steps:

1. Data pre-processing which transforms the raw source data into an appropriate form for the subsequent analysis (see section 2.1.2).

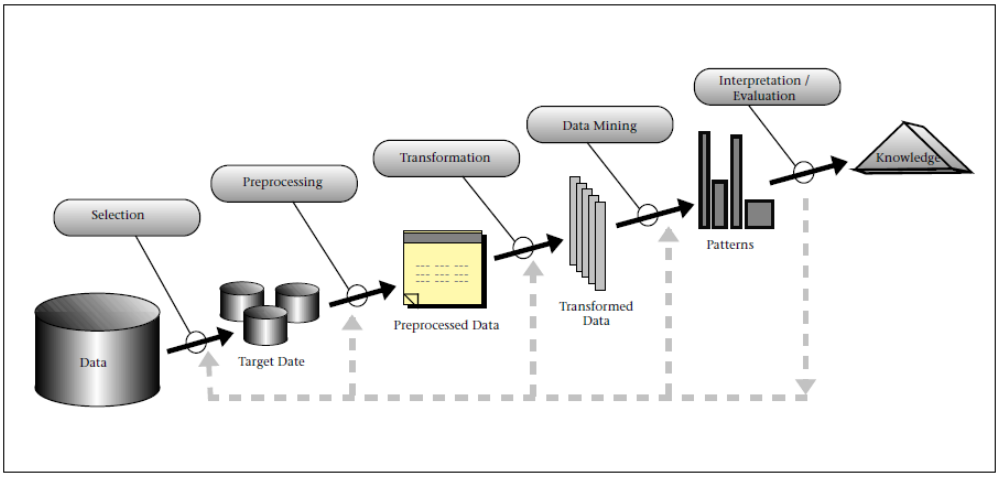

Figure 2.6: Process that are involved in discovering knowledge in databases (taken from: (FPSS96))

2. Data mining, which transforms the prepared data into patterns or models.

3. Post-processing of data mining results, which assess validity and usefulness of the extracted pattern and models, and presents interesting knowledge to the final users by using appropriate visual metaphor or integrating knowledge into decision support system.

2.3.2 Data Mining

Data mining is the process of automatically discovering useful information in large data repositories [**?**]. In practice, data mining has two primary goals, namely prediction and description [FPSS96]. Prediction involves some variables to predict unknown or future values of other variables of interest, while description focuses on finding human-interpretable pattern for data description. The boundary between these two goals is not very sharp; however knowing their difference is useful to understand the overall discovery goal.

To achieve data mining goals mentioned above, there are some methods available that can be applied to, such as [FPSS96]:

1. Classification: a function that maps (classifies) a data item into one of several predefined class.

2. Regression: a function that maps a data item to a real-valued prediction variable.

3. Clustering: a common descriptive task where one seeks to identify a finite set of categories or clusters to describe data.

4. Summarization: a function for finding a compact description for a subset of data.

5. Dependency modelling: a model that describes significant dependencies between variables.

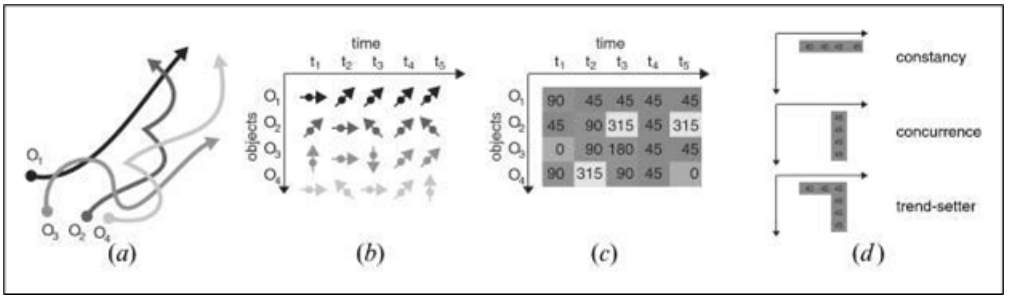

Figure 2.7: Steps in finding pattern based on Relative Motion analysis (taken from: (LvKI05))

Pattern in data mining has a description as: "a pattern is an expression E in some language L describing facts in a subset F_E of a set of facts F so that E is simpler than the enumeration of all facts in F_E" [FPSS96]. In other words, a pattern does not simply enumerate some facts but describes them all together as a whole [**?**].

There are many methods that can be applied to detect interesting pattern from trajectory. It can be done based on similarity in distance of motion parameters, i.e. similarity in route, lifespan, speed and direction [PKM+07] and similarity in relative movement pattern by defining relative movement matrix [LI02, LIW05]. Other approaches have also been proposed to deal with this subject by considering only the shape of trajectories, combining shape and temporal attributes [PKM+07], or applying spatial network constraint [TPN+09] as the parameter to calculate similarity in trajectories.

Among these approaches, relative motion patterns is considered to be the most appropriate ones for analyzing the iceberg trajectories. This is regarding to iceberg characteristic that has unconstrained network along its movements. The behaviour of iceberg can be analyzed through various factors, such as wind direction, ocean currents, sea surface temperature; which can be done by applying the concept of relative motion patterns.

2.3.3 Trajectory Similarity Based on Relative Motion Pattern

Laube in [LI02, LIW05, LvKI05] introduces analysis concept called REMO (RElative MOtion). The analysis is based on the comparison of motion attributes of point objects (e.g. speed, change of speed, or motion azimuth) over space and time, and also relates one object's motion to the motion of all others. The observation data (id, location and time) is transformed into a matrix which features a time axis, an object axis and motion parameters. Then, matrix is matched with the formalized patterns to reveal basic searchable relative movement patterns (see figure 2.7).

Figure 2.7 illustrates the construction of REMO analysis matrix. In figure 2.7(a) there are four moving objects with its respective trajectory. REMO pattern is analyzed by using motion parameter of each object on discrete time steps (t_1 to t_5). REMO matrix on figure 2.7(b) and 2.7(c) show that horizontal axis is defined as the temporal axis (time t), the vertical axis is defined as the object axis, and the values represent the motion parameter values (i.e. motion

azimuth). The temporal axis should be in ordered, while the object axis may not be in explicit order. To analyze the pattern, the motion parameters are grouped into discrete classes. E.g. motion azimuth is divided into eight classes (N, NE, E, SE, S, SW, W, NW) (see figure 2.7(b)) or it may also be expressed in code (figure 2.7(c)). From this matrix, REMO pattern can be derived by finding interrelations among objects that are clustered on the time-axis, across the objects, or combination of both (figure 2.7(d)).

In the REMO analysis, trajectories patterns can be identified and quantified based on user-defined motion parameters. There are two approach to analyze the pattern [LI02] :

- Motion Patterns without Neighbourhood Information: this is the basic relative motion patterns in the REMO analysis concept. Patterns are analyzed based on the order of time step and do not take spatial perspective into account. Thus, pattern is based on the order of motion parameters in time. There are three types of patterns in this group:
  - Patterns over time (horizontal axis)
  - Patterns across object (vertical axis)
  - Complex patterns over time and across objects (combination horizontal and vertical axis)
- Motion Patterns using Neighbourhood Information: this is an extend of the first pattern by considering the spatial constraints in REMO patterns, i.e. the motion of group objects in relation to the movement of other group members. The analysis of motion parameter can be based on [LvKI05]:
  - Spatial proximity: the proximity measurement that constrains the spatial extent of single object's motion pattern
  - Convergence: the convergence area that is defined based on some locations at certain time.

2.4 Summary

This chapter has illustrated the requirements that users have on iceberg data set and has also discussed two concepts of trajectory representation by considering the geometric and semantic properties. Research has identified there were four user requirements, i.e. data pre-processing, trajectory data extraction, event detection and similar pattern detection. Conceptually trajectory may be represented as design patterns, i.e. a flexible design which complexity depends to user requirements, or as data types, i.e. a generic data type which encapsulate trajectory attributes with its functionality. Further needs on trajectory also deal with data mining to get behaviour pattern of movement. This chapter has identified one approach of trajectory data mining which is based on the concept relative motion pattern. The relative motion pattern has provided two approach of pattern detections, i.e. by including and excluding the neighbourhood information.

Chapter 3

Trajectory Data Modelling

Data Modelling is a part of software engineering that describes how to represent and access data in the system. It is a critical step to define and analyze data requirements in a database. Within data modelling, basic information for database design, data structure and data integrity can be provided. For trajectory data modelling, the design of the data model has to combine spatial and temporal aspects of data, i.e. representation of the geometry of moving entities in time. This chapter highlights data models that are involved to represent trajectories in a database system. The first section (section 3.1) discusses conceptual data model, i.e. definition of entity and relationship among entities that have to be stored in a database. There are two kinds of data models, general and application-specific. Iceberg trajectory is used as a case study of application-specific model which is designed by extending general model to conform to the iceberg user requirements. The second section (section 3.2) describes the implementation of conceptual model, i.e. how trajectories are processed, by using Unified Modelling Language (UML) class diagram as a static representation of data model.

3.1 Conceptual Data Model

Conceptual data model is a description of data scope in the system. It concerns with how users see the information by emphasizing objects and rules that are involved in the real business world. Design of conceptual data model should be able to reflect the application requirements without the need for computer metaphors, i.e. complete and understandable design for users [TJ99].

For modelling trajectories, this research has applied the concept of trajectory design pattern. There are many formats of how to represent design pattern, each author may define his/her own style. The most common design pattern includes a definition of entity and the relationship among these entities.

An entity represents a collection of similar objects. It could be a collection of people, places, events, things or concepts. An entity has a similar analogy with class description in object oriented modelling. The main difference is on the scope of design. An entity may only provide information about entity's properties, while in class definition, data properties (attributes) has to be combined

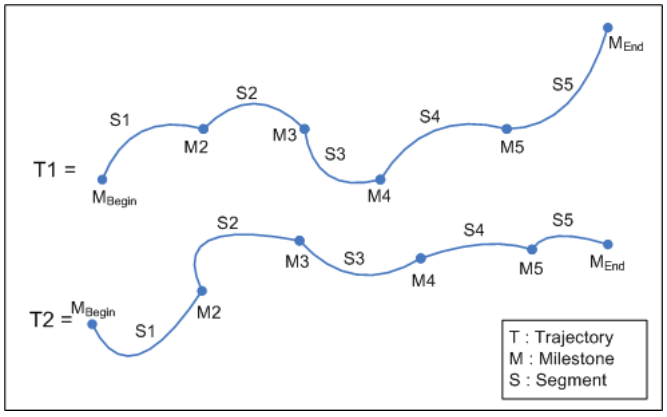

Figure 3.1: Trajectory definition as a sorted of entity's positions in time

with data behaviours (methods). In design pattern, an entity is represented only by its properties and is symbolized in a rectangle shape.

Relationships illustrate how two or more entities are related to each other. In design pattern, a relationship is described as a verb that shows the connection of two entities. A relationship may also be assigned with multiplicity, which is a range of allowable constraints between entities. It is symbolized as an ellipsoid circle.

3.1.1 General Data Model of Trajectory

Before representing trajectories in database system, the basic concepts of trajectory have to be presented. This research defines trajectory as a collection of entities position sorted in time. Record of an entity position in a time is termed as a milestone. Trajectory is modelled by using three type of entities which are formally defined as follow:

- Definition 1: Moving entity is a real world entity which location can be observed on the earth surface.
- Definition 2: Milestone is a record of moving entity which consists of position as a point and a time of recording.
- Definition 3: Trajectory is a sorted list of milestones. It consists of at least two milestones that represent as the beginning and the ending of an entity's trajectory.

The definitions above are illustrated in figure 3.1 and the general data model of trajectory can be seen on figure 3.2. Figure 3.2 shows that each moving entity may only moves on a trajectory and each trajectory may only belong to a moving entity. Trajectory consists of, at least two, milestones and each milestone may only exists on a trajectory.

The data model above is a general model that is applicable to any trajectory. For representing a specific application model, this model needs to be expanded to conform to application requirements. In the following section, the expansion

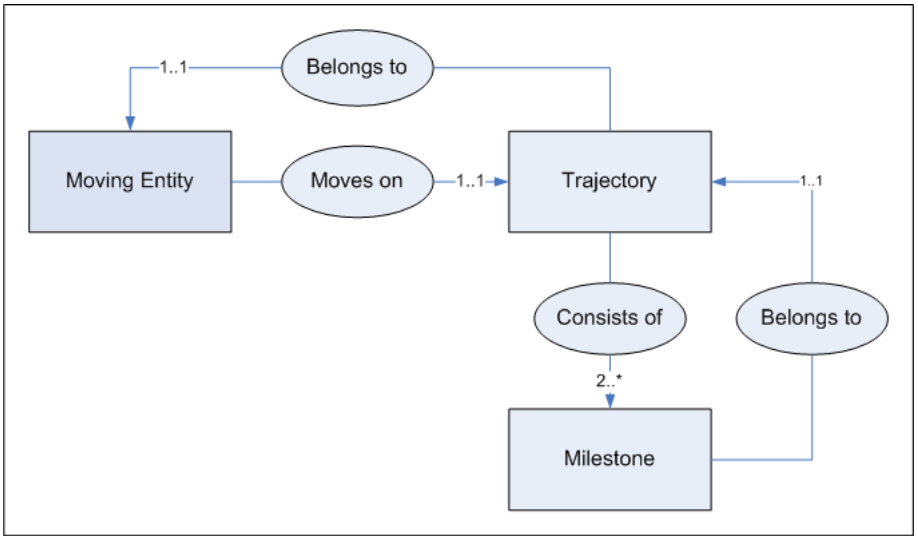

Figure 3.2: General trajectory model that consists of three main entities, named as Moving Entity, Trajectory, and Milestone

of general trajectory model is described with iceberg trajectory as the use case study.

3.1.2 Application-Specific Data Model

The expansion of a general model can be done by adding some new entity types or modifying the existing entities to fulfil the application requirements. Obviously, this expansion should be done based on analysis of user requirements.

Based on user requirement of iceberg movements (see section 2.1.2), the general trajectory model needs to be extended by modifying the characteristics of Milestone and adding some new entity types which are named *Event Detection* and *Similarity Pattern*. The illustration of conceptual iceberg data model is depicted on figure 3.3 and the following modification are introduced to the general model:

- *Iceberg* as a moving entity which trajectories are observed and analyzed. In this case, iceberg has the possibility to be split or calved into one or more smaller icebergs. If the calving event occurs on a milestone, it will create one or more new trajectories. Therefore, the constraint between class Milestone and Trajectory is modified by stating that one Milestone may belong to one-to-many Trajectory.

- *Event Detection* as a required entity to analyze the events that may occur to icebergs. There are various events that can be detected, such as calving or splitting, iceberg appearance or disappearance, grounded or floating and size reduction. Event_Detection is described by having relationship to Milestone and Iceberg since an occurred event can be observed through the difference of milestone's position and iceberg name.

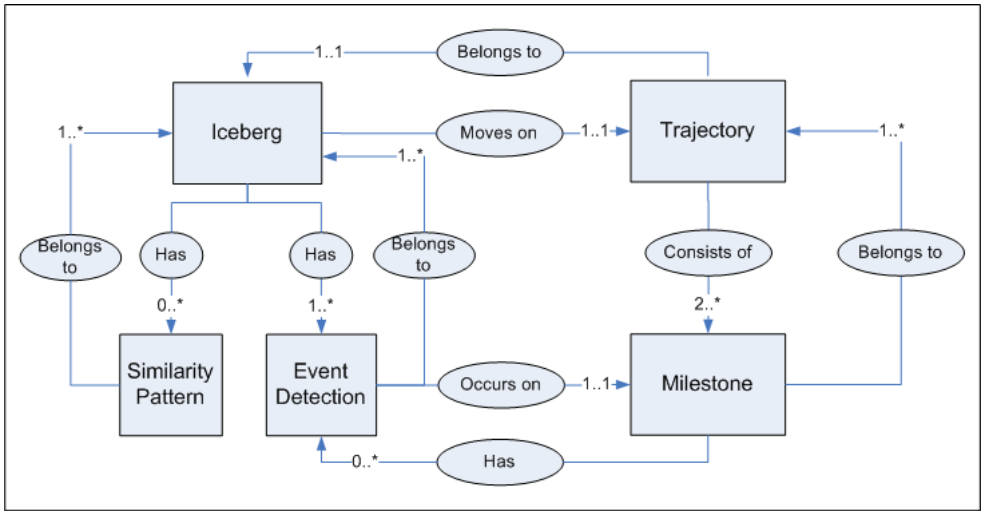

Figure 3.3: Application-Specific Data Model

- *Similarity Pattern* as a required entity to analyze the similarity movement of iceberg. The analyzed pattern is based on relative motion parameters, i.e. speed and direction. The output of this entity can be used to reveal the behaviour pattern of iceberg movement.

The two data models above are described on a conceptual level. These are high level designs which show the core entities that are involved in trajectory and the relationships among them. However, these models do not have detailed information on how they can be implemented. They still need to be provided with entity attributes and operations that can be used as guidance on how to process data among entities. One approach to describe the data models in more detail can be by using UML class diagram that is explained in the following section.

3.2 Trajectory Class Diagram

The next phase after designing conceptual design pattern is the implementation of data model. In this phase, data model should be able to present information and rules that are involved in trajectory. Trajectory design pattern needs to be provided in more detail. There are some alternatives on how to represent a complete data model. One approach that can be used is UML class diagram.

UML class diagram is a static diagram that describes a system structure together with its components. Class and the relationship between classes are the main components. A class is a template from which objects are created. It has attributes to store the value of object and method to manipulate the value. Class is typically modelled as a rectangle with three sections: the top section for the name of the class, the middle section for attributes declaration, and the bottom section for methods declaration.

In object oriented modeling, there are many kinds of relationship between

classes, such as association, generalization, aggregation, composition, association class and so forth. Below is a brief description of each relationship:

- Association is described as a general relationship type between classes. It implies that elements of two classes have a relationship. Normally, it is represented as a line, with each end connected to a class box.
- Generalization is used to indicate inheritance. It indicates that one of the two related classes (subclass) is considered to be a specialized form of the other (the super class). It is represented by a triangle shape on the end of the line on the super class and connects it to one or more subclass(es).
- Aggregation is an association that represents a part-whole or part-of relationship. It is used to show that a class is made up of other class. It is represented by a diamond shape.
- Composition is a more specific version of aggregation that is used to describe that components can be included in a maximum of one composition at a time. It is represented with a solid diamond shape.
- Association class is a construct that allows an association connection to have operations and attributes.

Before moving on to a further design, the author has defined a convention on how to name the entities that are involved in the data model. The goal of this convention is make the naming uniform, so database administrator or reader is able to differentiate directly the functionality of every name that is mentioned in the model. Here are the applied naming conventions:

1. Every word in a class name is written in italic and started with a capital letter. If class' name consists of two or more words, they are separated by an underscore. E.g. *Event_Detection*.
2. Every word in a class attribute name is written in italic and small letters. If attribute's name consists of two or more words, they are separated by an underscore. E.g. *ice_name*.
3. Every word in a class method name follows the same convention as attribute names, but method name is started with *tr* as the abbreviation of trajectory and ended by a pair of half circle bracket. E.g. *tr_travelled_time()*

3.2.1 General Trajectory Class Diagram

The class diagram of a general trajectory is represented in figure 3.4. Each component in this diagram is described as follows:

Milestone is a class representing a record of an entity's movement in a trajectory. It is characterized by having a triplet information of x and y position (it can be longitude-latitude or XY Cartesian coordinates) and time of when the position is recorded.

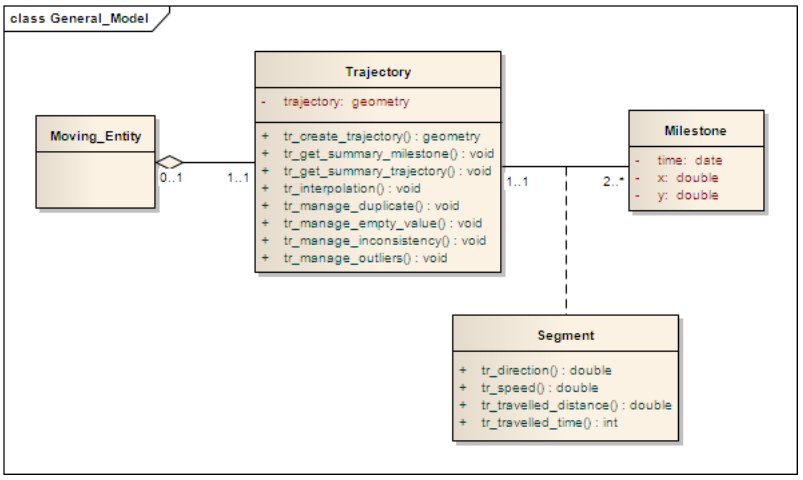

Figure 3.4: Class diagram of general trajectory

Trajectory is the central class of the diagram since it has aggregation relationship to *Moving_Entity* and binary association to *Milestone*. *Trajectory* is characterized with an attribute namely *trajectory* which stores the list value of milestone(s). This class has a constructor to create the geometry of trajectory (*tr_create_trajectory()*) and some manipulation methods for:

- Data pre-processing: *tr_manage_inconsistency()*, *tr_manage_duplicate()*, *tr_manage_outlier()* and *tr_manage_empty_value()*
- Summarizing the general information of trajectory during its lifespan: *tr_summary_trajectory()* and *tr_summary_milestone()*
- Interpolating data set that might be needed to guess a probable trajectory, especially for a data set that is recorded in irregular temporal resolution: *tr_interpolation()*

Segment is an associate class between *Trajectory* and *Milestone*. Segment consists of two consecutive milestones and collection of segments will create a trajectory. From this association, the following general information of trajectory can be extracted:

- Travelled time: *tr_travelled_time()*
- Travelled distance: *tr_travelled_distance()*
- Speed: *tr_speed()*
- Direction: *tr_direction()*

3.2.2 Iceberg Trajectory Class Diagram

Class diagram for iceberg trajectory is depicted on figure 3.5. Compare to the general class diagram on figure 3.4, there are some modifications have been made with respect to the needs of application.

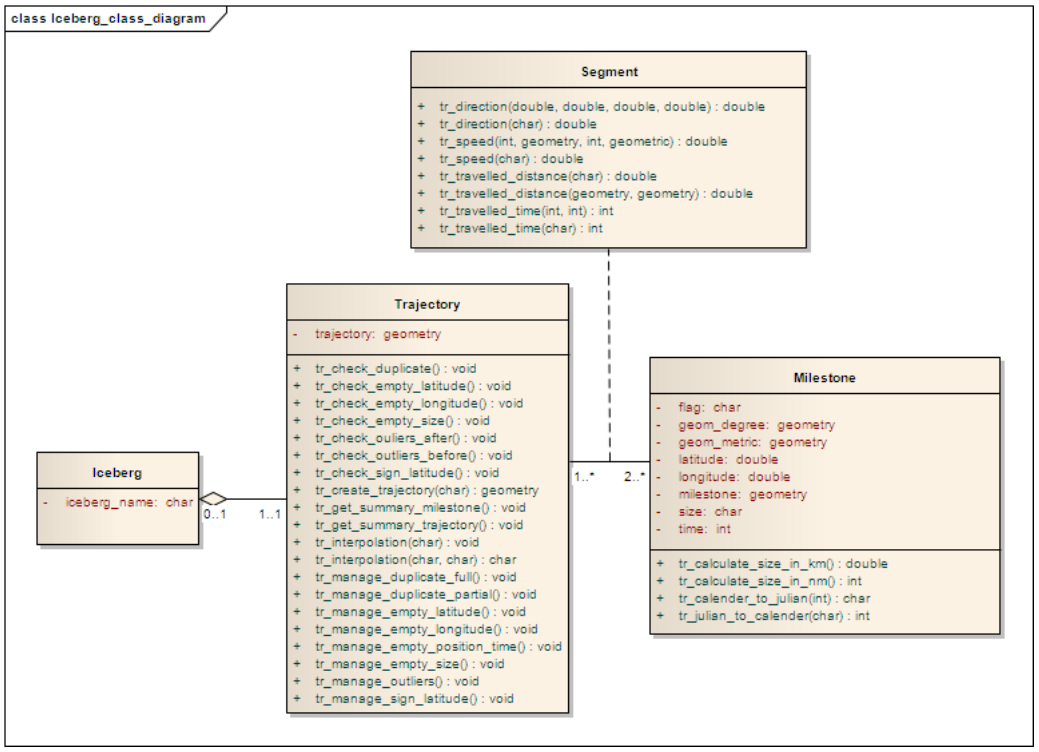

Figure 3.5: Class diagram of iceberg trajectory

There are some new attribute additions for *Milestone*. The data type of these new attributes follows as the original data source, i.e. iceberg name as character, longitude and latitude as double precision, time as integer and size as character. There is also an attribute termed as flag which is used to differentiate between an observed and a derived data of milestone. Milestone identified to have some data converter methods, i.e. to convert attributes data type into required data type on specific functions. This converter methods are listed as:

- *tr_calculate_size_in_nm()*
- *tr_calculate_size_in_km()*
- *tr_julian_to_calender()*
- *tr_calender_to_julian()*

In class *Trajectory*, there are some new data pre-processing methods have been added. These methods are intended to clean data set toward more specific attributes, such as to clean empty value of:

- Latitude: *tr_manage_empty_latitude()*
- Longitude: *tr_manage_empty_longitude()*
- Size: *tr_manage_empty_size()*

- Longitude, latitude and date: *tr_manage_empty_position_time()*

Also some methods to handle two types of duplication that are found in the iceberg data set:

- *tr_manage_duplicate_full()*
- *tr_manage_duplicate_partial()*

Class *Segment* has also been modified in terms of overloading its methods. Overloading is a form of polymorphism which enables one method name to have several functionality with different parameters and may also have different return type. These methods are defined to calculate travelled time, travelled distance, speed, and direction for along trajectory lifespan or along its segment. Below is an example of overloading method:

- *tr_travelled_time(char)*: is a method to calculate total travelled time of an iceberg during its lifespan.
- *tr_travelled_time(int,int)*: is a method to calculate travelled time between two consecutive milestones.

Based on application specific data model (see section 3.1.2), class diagram to represent iceberg trajectory also involves two required entities which are named as *Event_Detection* and *Similarity_Pattern*. For the sake of simplicity, this diagram is divided into two parts. The first part is dealing with class diagram for iceberg event detection and the second part is dealing with searching similarity pattern among iceberg movements.

Class Diagram of Event Detection

Class diagram shown on figure 3.6 reflects the structure for detecting events in a history of an iceberg. There are two supportive classes which are defined as *Iceberg_Event* and *Calving_Iceberg*. *Iceberg_Event* encapsulates all attributes and methods that are needed to classify each iceberg into one of the specified events, i.e. calving, grounded or floating. Result of classification is stored into *status_event*.

Calving_Iceberg has a role of detecting calving event with more specific functions, i.e. to define the parent of calving icebergs. When parent is detected, calving date and position can also be identified.

Class Diagram of Similarity Pattern

Figure 3.7 is related with searching similarity pattern among iceberg movements. The similarity search is based on the REMO concept (introduce earlier) that employs classification technique to find the behaviour patterns of iceberg movements. The search for patterns is based on two key features [LI02, LvKI05]:

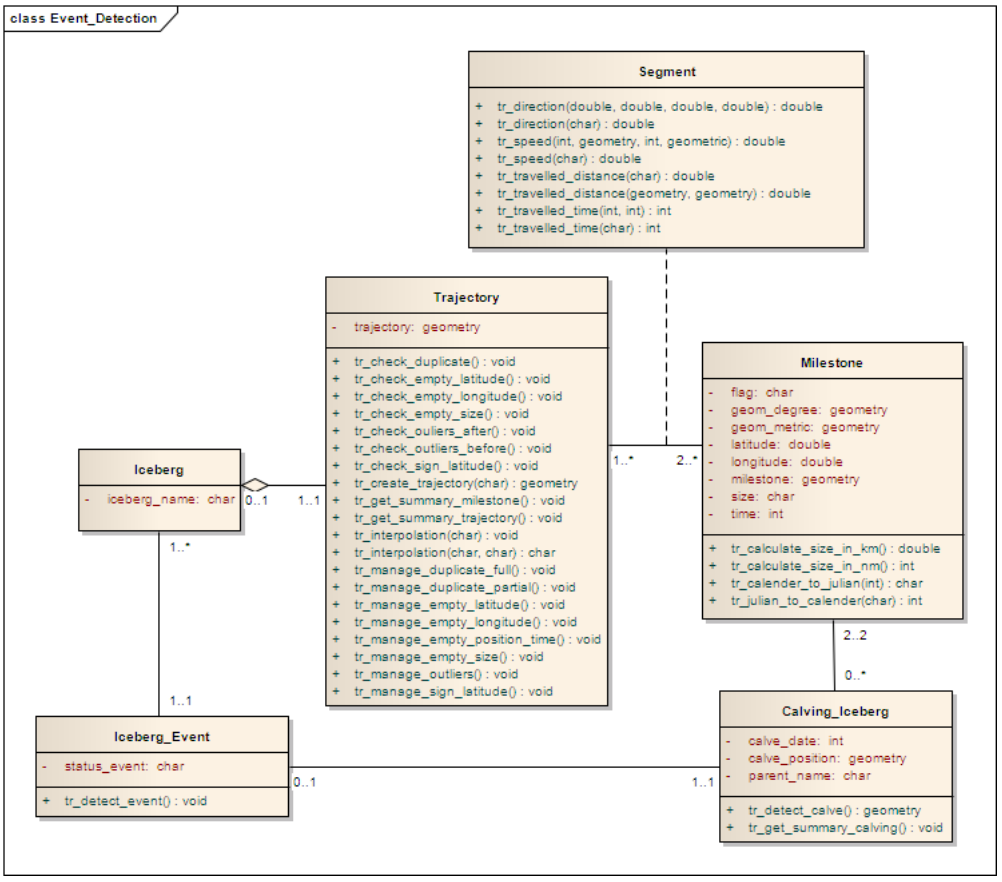

Figure 3.6: Class diagram for detecting event type that may occur on iceberg

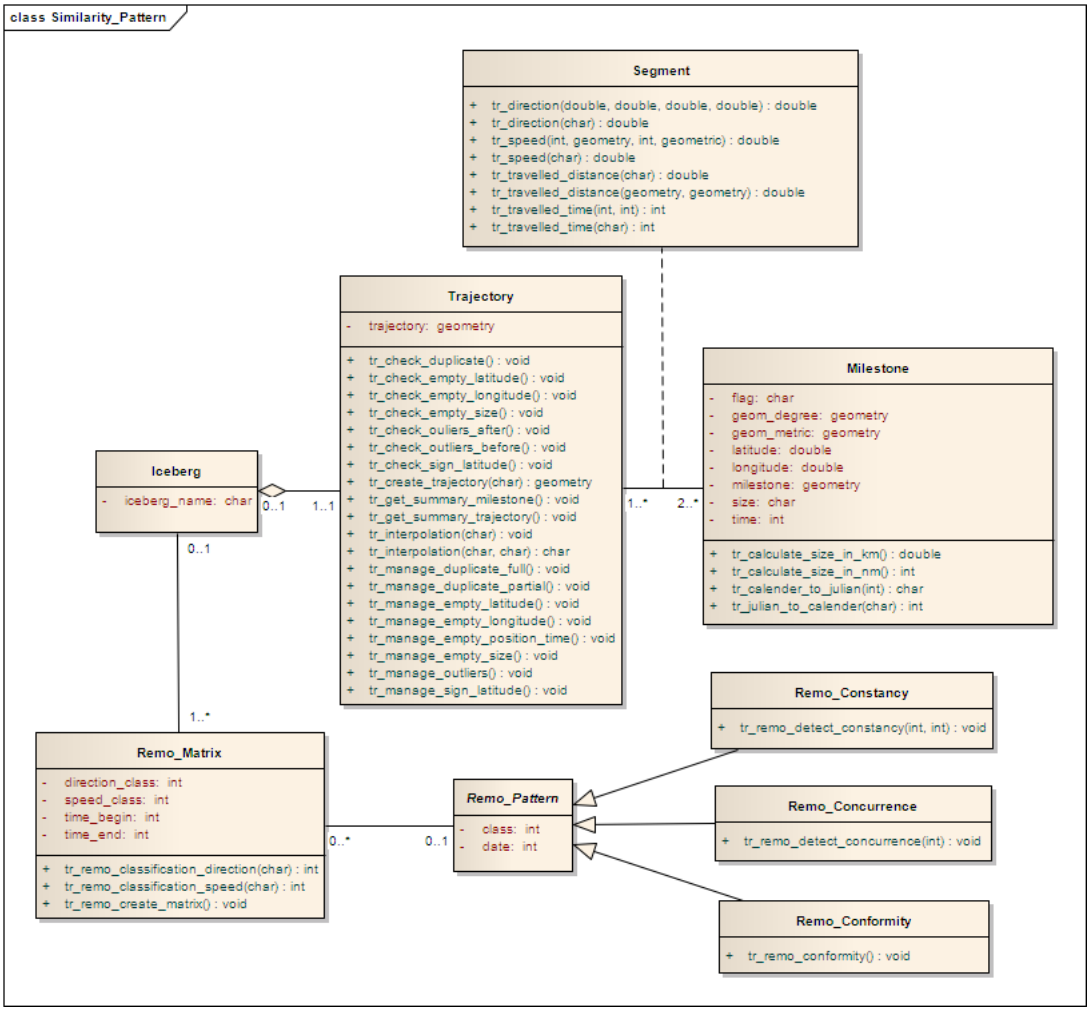

Figure 3.7: Class diagram for searching similar pattern of iceberg movement

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
O1	135	135	90	90	90	90	0	45	45	90
O2	90	45	45	0	0	45	0	0	0	0
O3	0	0	45	0	0	0	0	135	135	135
O4	90	135	135	135	90	0	0	0	135	135
O5	90	45	45	45	90	0	0	0	45	45

Figure 3.8: Matrix for searching behaviour pattern that consists of classified direction for five objects and ten time steps

1. Transformation of the trajectory data into matrix which features the classified parameters on regular time intervals.

2. Pattern detection.

Matrix is constructed as the list of icebergs' parameter in specified time steps. The matrix columns are based on regular time steps and the rows corresponds to icebergs' name (objects). The values of matrix are taken from classified parameters. As an example, figure 3.8 shows a matrix which holds classified direction of some icebergs on $time_1$ to $time_{10}$.

Pattern can also be termed as a search template. There are three types of patterns that can be extracted from matrix. These patterns are named as:

1. Constancy for a search template that spans over time, i.e. one object over several times (see figure 3.9a).

2. Concurrence for a search template that moves across objects. i.e. several objects at one time (see figure 3.9b).

3. Conformity for a search template that combines the search over times and across objects, i.e. several objects at several times (see figure 3.9c).

These search templates are illustrated in figure 3.9. Referring to the key features and pattern types mentioned above, in the class diagram of *Similarity_Pattern* there is a class termed as *Remo_Matrix*. This class encapsulates method to generate matrix based on the specified time ranges (from *time_begin* to *time_end*) and stores the classified parameter into an attribute. This research applies speed and direction as motion parameters, hence the classified parameters are stored into *speed_class* and *direction_class*.

Pattern detection is represented by an abstract class of *Remo_Pattern*. This class has attributes of the classified parameter (*class*) and the date (*date*) which can be inherited by three sub-classes, i.e. *Remo_Constancy*, *Remo_Concurrence* and *Remo_Confirmity* to store the identified patterns.

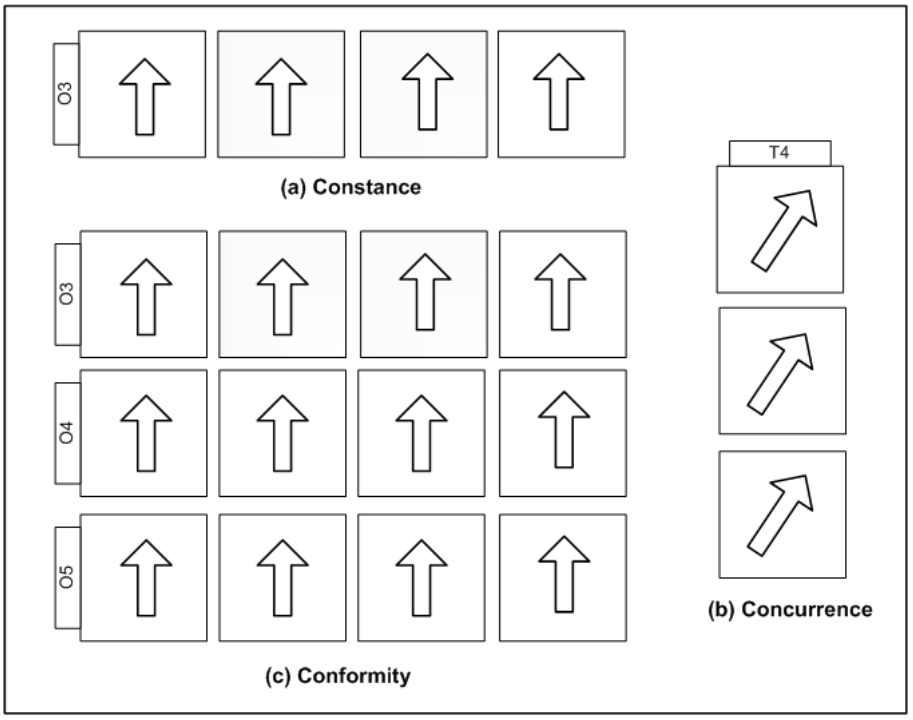

Figure 3.9: Three types of similarity pattern

3.3 Summary

This chapter has discussed general and application-specific trajectory data models. The basic models was based on trajectory design pattern which illustrated the required entities and relationship. There were three general entities involved: moving entity, trajectory and milestone. To conform with application requirements, the general model has been modified by adding two new entities, namely event detection and similarity pattern. Afterwards, the data models were transformed into UML class diagram. There were two class diagrams based on user requirements, i.e. class diagram for detecting the iceberg events and the similar patterns. Iceberg event detection has divided into two subclasses, i.e. to classify three type of events, i.e. calving, grounded and floating; and to detect the parent, position and time of calving occurrence. In similarity pattern class diagram, it showed data structure to classified the iceberg parameters into three type of patterns, i.e. constancy (pattern over times), concurrence (pattern across objects) and conformity (pattern over times and across objects).

Chapter 4

Trajectory Back End Support Implementation

This chapter presents the implementation of trajectory data modelling. The focus of the discussion lies in the four groups of database functions. The first group, back end support for data pre-processing, is discussed in section 4.1. Data pre-processing involves functions for cleaning data set from inconsistencies, empty values, data duplicates and outliers. Section 4.2 describes the second group of back end support for extracting the general information of trajectory. Each trajectory is characterized by having a travelled time, travelled distance, speed and direction. Section 4.3 discusses the third back end support for iceberg event detection which can also be applied for identifying parent child relationship among icebergs. This chapter is finalized by implementing trajectory data mining techniques for searching similar patterns of iceberg movements (section 4.4).

All functions are implemented by programming language PostgreSQL, i.e. PL/pgSQL and using PostGIS functions. Some functions are declared by using SQL cursor. A SQL cursor in PostgreSQL is a read-only pointer that has a fully access to execute SELECT statements. Cursor can be used to manage which rows should be retrieved/fetched from data set.

4.1 Back End Support for Data Pre-processing

Data pre-processing is a preliminary process that has to be done before computing data in further process. For iceberg data set, data pre-processing consists of four steps, i.e. data integration, data reduction, data cleaning, and data transformation. The sequence activity of data pre-processing can be seen on figure 4.1.

1. **Data integration** involves activity to integrate all data sources into one working space. The Antarctic iceberg data set consists of four different spread sheet files that represent iceberg movements in each quadrant. To analyze the phenomena in the whole part of Antarctica, these spread sheet files have been integrated into one working table.

2. **Data reduction** is dealing with selection of the main attributes from data source. Compares to the original data set that consists of eight attributes, i.e. iceberg name, date, longitude, longitude sign, latitude, latitude sign, size and image source; this research only uses five attributes that are considered as the basic information of the iceberg trajectory. These attributes are iceberg name, longitude, latitude, date and size.

3. **Data cleaning** is related with cleaning data from errors that are found on iceberg data set. Initial inspection has found that data inconsistencies, empty values, data duplicates, typing errors and outliers are errors that should be removed or managed in the data set.

4. **Data transformation** is dealing with transforming some attributes into appropriate data type that is needed in further process. In spatially-enabled database, transformation data into geometry type should also identify the projection system that is referred to, i.e. the SRID (Spatial Reference System Identifier). This research refers to WGS84 (SRID = 4326) for the geometry type in degree unit and Antarctic Polar Stereographic (SRID = 3031) for the geometry type in metric unit. For trajectory analysis purpose, the three components of milestone, i.e. longitude, latitude and date; is transformed into 3D geometry point.

There are two types of functions for data pre-processing. These functions are applied for checking and managing data from inconsistencies, empty values, duplicates and outliers. These functions are implemented based on the constraints below:

This section is focused on the implementation of functions for data cleaning. Functions for managing data from errors are implemented based on the constraints below:

1. Milestone may not have empty value in position and time.

2. Latitude may not be assigned in positive value with respect to NIC requirement that all the Antarctic icebergs should be located on the south of 60° south latitude.

3. The speed of every two consecutive milestones may not be exceeded 80 km/day.

The list of functions for data pre-processing can be seen on table 4.1 and the implementation on these functions have been attached on appendix A.

4.1.1 Data Consistency Management

Data consistency management is applied to attributes of iceberg name and latitude attributes. In the original iceberg data set, name of the iceberg has been a mixed up between upper and lowercase letters. E.g. some icebergs are named as A20A, a20A, A20a and a20a. There is no difference among these names, hence all icebergs' name have been updated into uppercase letter(s).

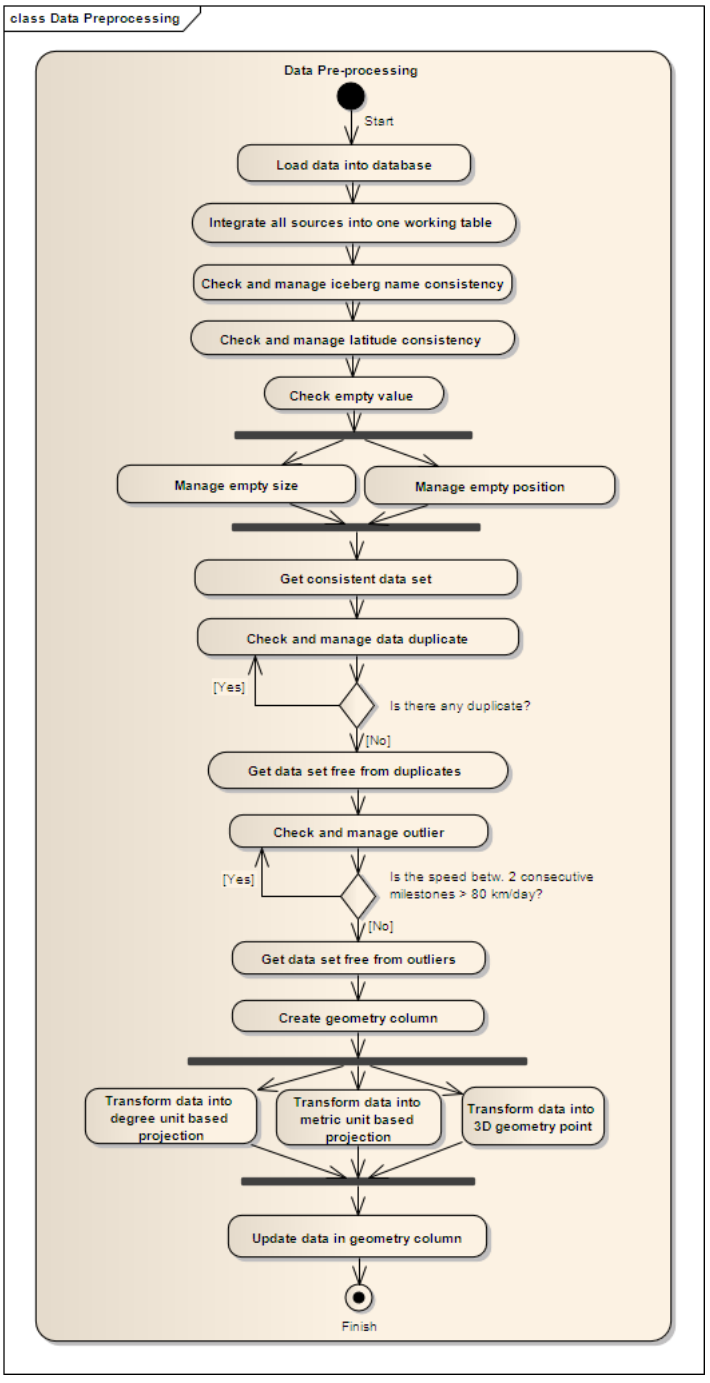

Figure 4.1: Activity diagram of data pre-processing which consists of data integration, data reduction, data cleaning, and data transformation

Table 4.1: List of Function for Data Pre-processing

Name of Function	Parameter Input	Return Type
tr_data_integration	none	setof table
tr_manage_sign_latitude	none	setof table
tr_manage_empty_latitude	none	setof table
tr_manage_empty_longitude	none	setof table
tr_manage_empty_size	none	setof table
tr_manage_empty_position_time	none	setof table
tr_manage_duplicate_full	none	setof table
tr_manage_duplicate_partial	none	setof table
tr_manage_outliers	none	setof table

Function to manage the sign of latitude is applied for milestone that has latitude as positive value. This is regarding to the second constraint above that states the Antarctic iceberg can not be positioned on the north of equator. Therefore for all positive latitudes are assigned as negative ones.

4.1.2 Empty Value Management

Empty value management is applied to milestone that has no value either in its latitude, longitude, size or empty in both position and date attributes.

Empty value in longitude and/or latitude position longitude and/or latitude has been handled by applying a simple linear interpolation. Linear interpolation is defined using the following equations [NRRT05]:

$$(x - x_1)(t_2 - t_1) - (x_2 - x_1)(t - t_1) = 0 \tag{4.1}$$

$$(y - y_1)(t_2 - t_1) - (y_2 - y_1)(t - t_1) = 0 \tag{4.2}$$

In these equation, x and y refer to milestone that has empty value either in its longitude (x) and/or latitude (y). Therefore to interpolate the missing longitude (x), function *tr_manage_empty_longitude()* uses equation 4.1. It is done by considering x_1 as the longitude of the previous milestone and x_2 refer to the longitude value of the next milestone of x (see algorithm 1). The same mechanism applies for managing latitude by using equation 4.2.

Management empty sizes has been implemented by taking the size value of the previous milestone. While if a milestone has empty value in both position and date, it is assumed to be a non-valid milestone that will be removed from database.

4.1.3 Data Duplicate Management

Data duplicate is defined based on four attributes, i.e. iceberg name, longitude, latitude and date. The iceberg data set shows there are two type of duplications, i.e. full and partial. A milestone is defined having a full duplication if other

Algorithm 1: Manage Empty Longitude

Ensure: Working table has been ordered by iceberg name and date
1: Open cursor on table
2: Fetch three milestones (m_1, m_2, m_3)
3: **loop**
4: **if** $m_1.name = m_2.name$ **and** $m_2.name = m_3.name$ **then**
5: **if** $m_2.longitude = 0$ **then**
6: $m_2.longitude \leftarrow ((m_3.longitude - m_1.longitude)/(m_3.date - m_1.date)) + m_1.longitude$
7: update table for m_2
8: **end if**
9: $m_1 \leftarrow m_2$
10: $m_2 \leftarrow m_3$
11: fetch one milestone as m_3
12: **end if**
13: **end loop**
14: close cursor

milestone(s) exists with similar value in these four attributes. If the similarity is found only on iceberg name and date, milestone is defined having a partial duplication. Hence for managing data duplicates, there are two steps needed:

- The first step is dealing with full duplication. This is done by taking only one milestone for each group of milestone that have full duplication (see algorithm 2).
- The second step is applied for partial duplication which has different longitude and latitude on its duplicate group members. A similar approach as the previous step, this step also takes only one representative of duplicate milestones. Longitude and latitude is assigned as the average of all longitude and latitude values in that group respectively (see algorithm 3). This step able to clean data set from partial duplication; however it has a limitation when there is outlier(s) in the group. This limitation can be avoided by managing the outliers beforehand.

4.1.4 Outliers Management

Outliers management has been applied for milestones that have speed more than 80 km/day [Ra80]. Speed is calculated between two consecutive milestones of the same iceberg. There are two functions that have been implemented to analyze pre- and post-processing of handling outliers. Function of pre-processing outliers shows that outliers exist as the result of:

1. Typing errors which cause the incidental exchange of plus and minus sign and doubled value.
2. Switch of values between longitude and latitude.

Algorithm 2: Manage Full Duplication

Ensure: Working table has been ordered by iceberg name and date
Ensure: Table to store result ($free\_full\_duplicate$) has been created
1: Open cursor on table
2: Fetch two milestones (m_1, m_2)
3: **loop**
4: **if not** ($m_1.name = m_2.name$ **and** $m_1.date = m_2.date$ **and** $m_1.longitude = m_2.longitude$ **and** $m_1.latitude = m_2.latitude$) **then**
5: store m_1 into table $free\_full\_duplicate$
6: **end if**
7: $m_1 \leftarrow m_2$
8: fetch one milestone as m_2
9: **end loop**
10: close cursor

Algorithm 3: Manage Partial Duplication

Ensure: Working table has been ordered by iceberg name and date
Ensure: Table to store result ($free\_partial\_duplicate$) has been created
1: group milestone by iceberg name and date
2: count member for each group
3: **loop**
4: **if** number of group = 1 **then**
5: store milestone into table $free\_partial\_duplicate$
6: **else**
7: $new\_longitude \leftarrow average(longitude)$
8: $new\_latitude \leftarrow average(latitude)$
9: take one milestone and assign $new\_longitude$ and $new\_latitude$ as its longitude and latitude
10: store milestone into table $free\_partial\_duplicate$
11: **end if**
12: **end loop**

3. Data anomaly which is related with the actual value of record, e.g. in a short period of time there is a large movement in space.

Function to manage the outliers is defined for handling the typing errors, i.e. the consistency of longitude value. It works by comparing the consistency of current longitude with its previous record. This function is implemented based on algorithm 4.

Algorithm 4: Manage Outliers

Ensure: Table to store the clean data set from outliers ($free\_outliers$) has been created
Require: The first milestone has hold the right value
1: Retrieve two consecutive milestone (m_1 and m_2) of the same iceberg
2: Calculate speed between m_1 and m_2
3: **if** speed > 80 **then**
4: Create temporary milestone ($temp$)
5: $dif \leftarrow m_1.longitude - m_2.longitude$
6: **if** $dif >= 10$ **then**
7: $temp.longitude \leftarrow m_2.longitude/10$
8: **else if** ($m_1.longitude > 0$ **and** $m_2.longitude < 0$) **or** ($m_1.longitude < 0$ **and** $m_2.longitude > 0$) **then**
9: $temp.longitude \leftarrow m_2.longitude * -1$
10: **end if**
11: $temp.longitude \leftarrow m_2.longitude$
12: Calculate speed between $m1$ and $temp$
13: **if** speed < 80 **then**
14: Insert into $free\_outliers$
15: **end if**
16: **end if**
17: Insert into $free\_outliers$

Function of post-processing shows that there are still some outliers that occur due to switch values. An approach to handle this problem is done by having human intervention, i.e. manual editing.

Figure 4.2 and 4.3 illustrates the comparison before and after outliers are managed in database.

Below is a summary of data pre-processing implementation in terms of number of error records that have been identified and corrected from 301 icebergs with 15737 records:

- Full duplication (identical value in all attributes: iceberg name, date of recording, latitude, longitude and size): 251 records.
- Partial duplication (identical value in iceberg name and date of recording): 300 records.
- Empty value in latitude, longitude and date: 5 records.

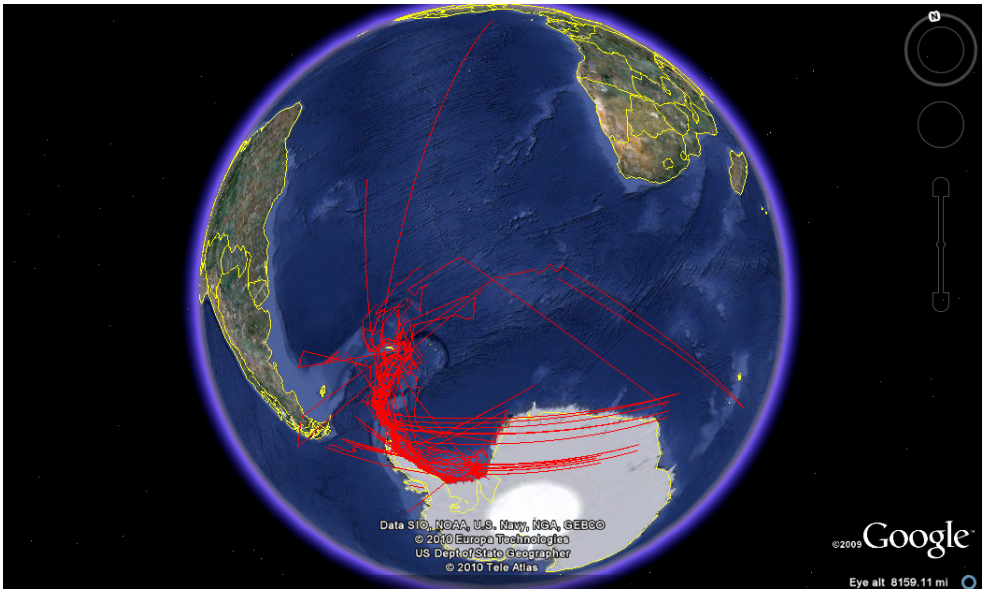

Figure 4.2: Trajectory with outliers

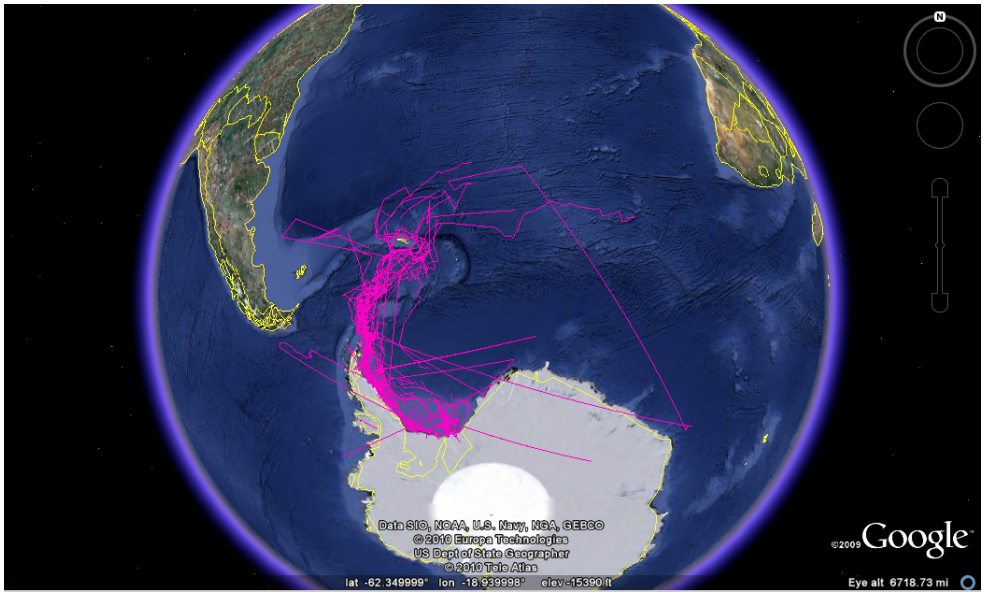

Figure 4.3: Trajectory that has been cleaned from outliers

- Empty value only in longitude: 7 records.
- Empty value only in latitude: 6 records.
- Empty value in size: 12 records.
- Incidental exchange of plus and minus sign for latitude: 57 records.
- Outliers (milestone that has the speed value more than 80 km/day): 122 pairs of records.

4.2 Back End Support for Trajectory Data Extraction

The step for extracting the general information of trajectory is shown on figure 4.4. The involved functions are dealing with:

- Converting some attributes into required data type.
- Calculating geometric properties of trajectory, i.e. travelled time, travelled distance, speed, and moving direction.
- Creating trajectory from the list of milestone.
- Aggregating all of this information as summary of iceberg movement.

Table 4.2 shows the list of implemented functions for extracting data from trajectory and the actual source code of these functions has been attached on appendix B.

4.2.1 Data Conversion

Functions for data converter are dealing with converting size and date into required data type. Data converter is needed with respect to other process, such as calving detection that needs to compare the size of two iceberg and analysis of similarity pattern that needs input of time range from user in calender (Gregorian) format.

Original data data set store size as five characters. E.g size is declared as ”45x38”. It shows the length and the width of iceberg in Nautical Miles (nm). For computation purpose, size has to be casted into numerical type. There are two functions have been implemented to calculate size in nm and km (kilometer) unit.

Iceberg date has its own specific format. It consists of seven digits in which the first four digit shows the year and the last three digit shows the order of the day in that year. E.g. iceberg A01 has a record 1978299, which means that iceberg was recorded in the year 1978 on the 299^{th} day. This format enables user, such as system developer, to apply some mathematical operations. However on presentation level, this is not an appropriate form to be used. Therefore, there are two functions have been implemented to convert iceberg date into Gregorian format and vice versa.

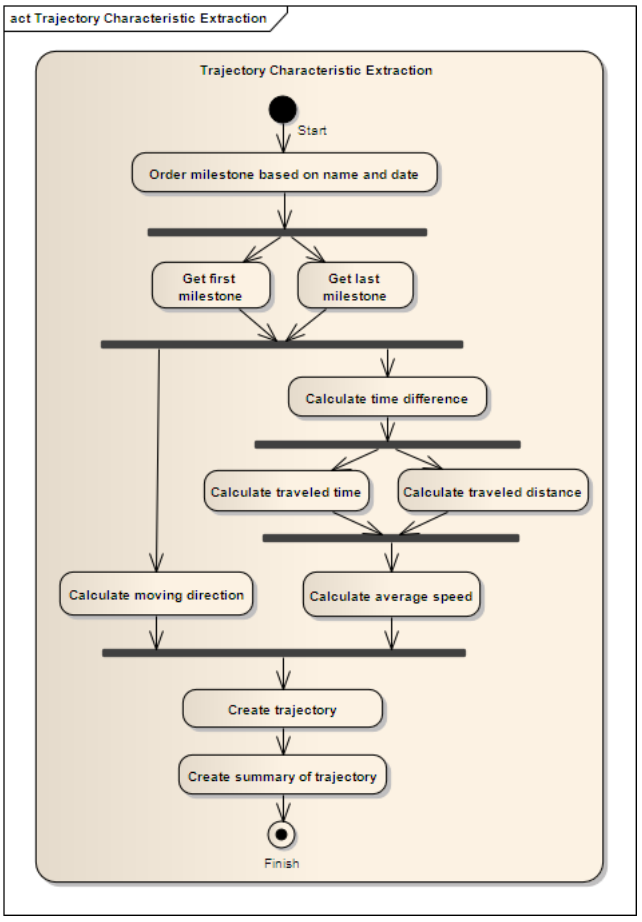

Figure 4.4: Activity diagram of extracting general information from trajectory

Table 4.2: List of Function for Trajectory Data Extraction

Name of Function	**Parameter Input**	**Return Type**
tr_create_trajectory	ice_name: text	geometry
tr_interpolation	ice_name: text	setof table
tr_interpolation	time_from: text, time_to: text	text
tr_direction	ice_name: char	double
tr_direction	point1: geometry, point2: geometry	double
tr_speed	ice_name: char	double
tr_speed	date1: int, point1: geometry, date2: int, point2: geometry	double
tr_travelled_distance	ice_name: char	double
tr_travelled_distance	point1: geometry, point2: geometry	double
tr_travelled_time	ice_name: char	integer
tr_time_difference	date1: int, date2: int	integer
tr_is_leap_year	date: int	boolean
tr_calender_to_julian	date: text	integer
tr_julian_to_calender	date: int	text
tr_calculate_size_in_nm	size: text	integer
tr_calculate_size_in_km	size: text	double
tr_num_milestone	ice_name: char	integer
tr_get_summary_trajectory	none	setof table
tr_get_summary_milestone	none	setof table

4.2.2 Trajectory Characteristics Extraction

Trajectory is characterized of having a travelled time, travelled distance, speed, and direction. There are four main functions have been implemented as overloading methods to extract these characteristics. These functions perform several different functionalities depending on the number and the type of parameters. Calculation can be done either for the whole lifespan of a trajectory or between two consecutive milestones (in one segment).

Travelled time is calculated as a number of days an iceberg has travelled. Travelled time in one segment can be considered as the time difference between two dates. While for the whole lifespan of a trajectory, it is calculated as the total days of each segment. Function to calculate travelled time has been implemented with support of two other functions, i.e. *tr_time_difference()* and *tr_is_leap_year()*. Algorithm 5 to calculate travelled time.

Algorithm 5: Calculate Travelled Time

Ensure: Table reference has been ordered by date
Require: Iceberg name (ice_name)
1: Open cursor on table reference where iceberg name = ice_name
2: Fetch two milestones (m_1, m_2)
3: $ave\_time = 0$
4: **loop**
5: $dif \leftarrow (m_2.date - m_1.date)$
6: $ave\_time \leftarrow ave\_time + dif$
7: $m_1 \leftarrow m_2$
8: fetch one milestone as m_2
9: **end loop**
10: close cursor
11: **return** ave_time

Travelled distance is calculated as a Cartesian distance between two points. Function to extract this value is implemented by using PostGIS built-in function *st_distance()*. Parameter input of this function should be in metric unit geometry. Therefore to calculate travelled distance, longitude and latitude attributes have to be converted into point then transformed into metric unit based on the Antarctic Polar Projection System.

Speed calculation is based on the value of travelled distance and travelled time. Speed can be calculated as an average speed along a trajectory or speed between two consecutive milestones. For average speed, the calculation only consider the distance between the first and the last milestone divided by travelled time along trajectory.

The same condition for extracting direction of a trajectory. It can be calculated as major direction of a trajectory, which considers the first and the last milestone, or direction of two consecutive milestones. Function to calculate direction employs PostGIS built-in function *st_azimuth()* which require two metric unit geometries as parameter inputs.

The output of function *st_azimuth()* is in radian unit (-3.14 to +3.14). Com-

mon direction is presented in degree unit (0° to 360° or -180° to +180°). Hence in the implemented function, data is converted into degree unit which is based on the equation 4.3:

$$degree = radian * (180/3.14); \tag{4.3}$$

4.2.3 Creation of Trajectory

Trajectory is created by using function *tr_create_trajectory()*. This function is implemented by putting all milestones of the same iceberg into an array. Afterward, array is transformed into a linestring by applying PostGIS function *st_makeline_garray(geometry[])*. Algorithm 6 shows the step of creating trajectory.

Algorithm 6: Create Trajectory

Ensure: Table reference has been ordered by date
Ensure: Table reference has 3D geometry point ($pointm$)
Require: Iceberg name (ice_name)
1: Open cursor on table where iceberg name = ice_name
2: Fetch one milestones (m_1)
3: **loop**
4: store $m_1.pointm$ into array
5: fetch one milestone as m_1
6: **end loop**
7: **if** array length > 1 **then**
8: $trajectory \leftarrow st\_makeline\_garray(array)$ {create trajectory as a linestring}
9: **end if**
10: close cursor
11: **return** $trajectory$

A milestone is represented as a 3D geometry point. Research uses PostGIS function *st_makepoint(double, double, double)* to create this. It involves three attributes of a milestone, i.e. longitude, latitude and date; as parameter functions.

Figure 4.5 illustrates trajectory of some icebergs that occur from 1978 to December 1984. In this figure, trajectory has been described as a linestring of some milestones.

4.2.4 Trajectory Summarization

There are two functions have been implemented for providing summary information of trajectory:

- Function of (*tr_get_summary_trajectory()*) provides summary of trajectory characteristics. This function can be used to generate total travelled time, total travelled distance, average speed, major direction and number of

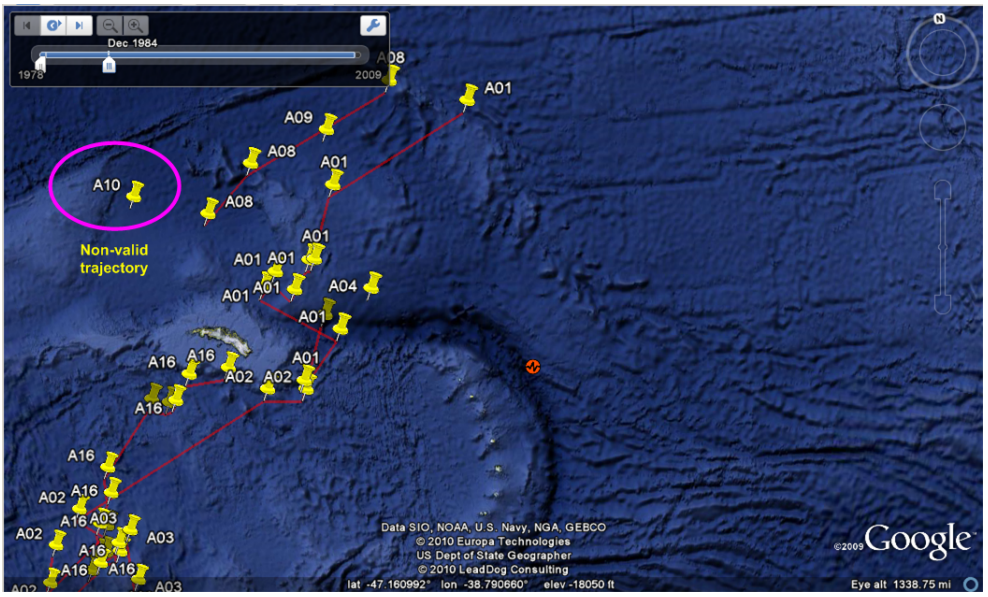

Figure 4.5: Visualization of trajectory in Google Earth from 1978 to 1984

	iceberg character var	traveled_time integer	traveled_dist double precision	ave_speed double precision	direction double precision	trajectory geometry	num_milestone integer
41	A29	769	3112.15871137349	4.04702043091481	95.4316728960279	0102000040040	4
42	A30	57	311.664835025071	5.46780412324686	31.0856632272271	0102000040030	3
43	A31	308	424.215016873924	1.3773214833569	34.4275508357103	0102000040080	8
44	A32	84	128.849681873906	1.53392478421316	-33.2176743924661	0102000040060	6
45	A32A	186	1257.74579515412	6.76207416749529	-5.88427013069607	0102000400FC	15
46	A32B					0101000409A9	1
47	A33	30	0	0	0	0102000040020	2
48	A34	53	2472.29013304573	46.6469836423722	74.7626357572029	0102000040020	3
49	A35	656	877.298431391051	1.33734516980343	-167.847029826624	0102000402D0	45
50	A35A	772	1593.39217336701	2.06397949918007	-66.106479549975	0102000402A0	42

Figure 4.6: Summary of trajectory

milestones of each icebergs. From this summary, user may analyze the the statistic of these characteristics. E.g. icebergs in quadrant A have average lifetime about 825 days, average travelled distance 3598 km and average speed 6.5 km/day.

Furthermore, this summary also shows some non-valid trajectories, i.e. iceberg that has only one milestone. This iceberg has to be removed from database since violates trajectory definition (see section 3.1.1).

Figure 4.6 shows the snapshot of table summary trajectory and figure 4.5 shows one non-valid trajectory, i.e. A10, that should be removed from a database.

- *tr_get_summary_milestone()* is related with summary of the first and the last milestone with respect to it position, date and size of each trajectory.

Table 4.3: List of Function for Event Detection

Name of Function	Parameter Input	Return Type
tr_detect_event	none	setof table
tr_detect_calve	none	setof table
tr_get_summary_calving	none	setof table

4.3 Back End Support for Event Detection

Back end for event detection is implemented to classify events that occur during iceberg lifespan. There are three types of event: calving, grounded and floating. In general these events are differentiated based on the following constraints:

- Calving is defined for an iceberg that has similar main name to other icebergs. E.g. the group of iceberg A60 (see figure 2.2) consists of three icebergs: A60, A60A and A60B. These icebergs are classified as calving. The leader of the group, i.e. iceberg that has no suffix in its name, is assumed as a parent and others are assumed as child.

- Any iceberg can be classified either as grounded or floating. If the travelled distance during iceberg lifespan is zero, it is assumed to be grounded; otherwise it is assumed to be floating.

The step for defining the general iceberg event and calving detection is illustrated on figure 4.7. Table 4.3 shows the list of functions for this purpose. These functions have been applied on icebergs of the quadrant A of Antarctica. In total there are 95 of icebergs. Among this number, 48 were classified as calving icebergs, 8 grounded icebergs and 36 floating icebergs. From 48 calving icebergs, 12 icebergs are assumed as parent and the rest are classified as child.

Further process on calving detection is needed to define the source of calving, i.e. iceberg's parent, the site and time of occurrence. In general, calving event can be detected from iceberg name. However, the current naming system that has been used by National Ice Center (NIC) (see section 2.1.1) creates difficulty for identifying these needed information.

This difficulty is illustrated in figure 4.8. This figure shows there is one group of iceberg which consists of A60, A60A, A60B and A60C. Since these names are based on alphabetical order, it can be defined directly that:

- A60 is the root of the series

- A60A is calved from A60.

However, this approach can't be applied to the icebergs A60B and A60C or to any other further descendants. There are possibilities that may have occurred to the actual calving events on this group.

- A60B may be calved either from A60 or A60A.

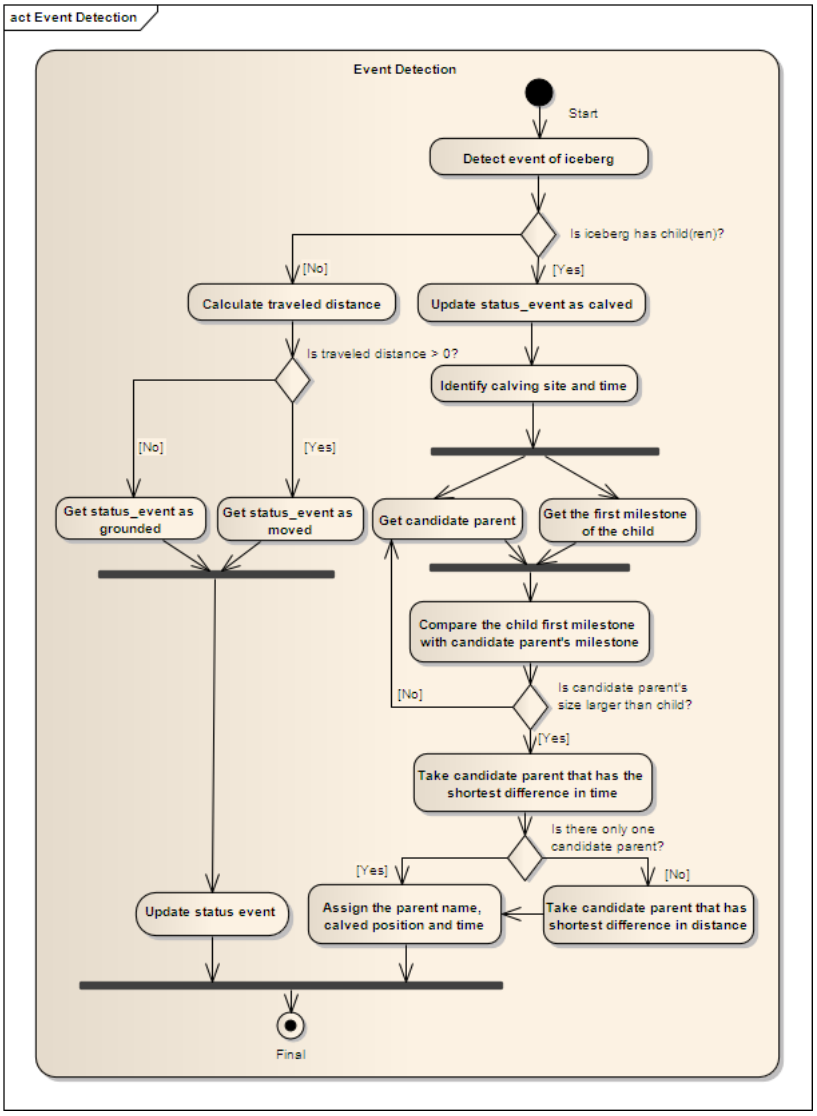

Figure 4.7: Activity diagram of detecting three type of iceberg event, i.e. calving, grounded and floating

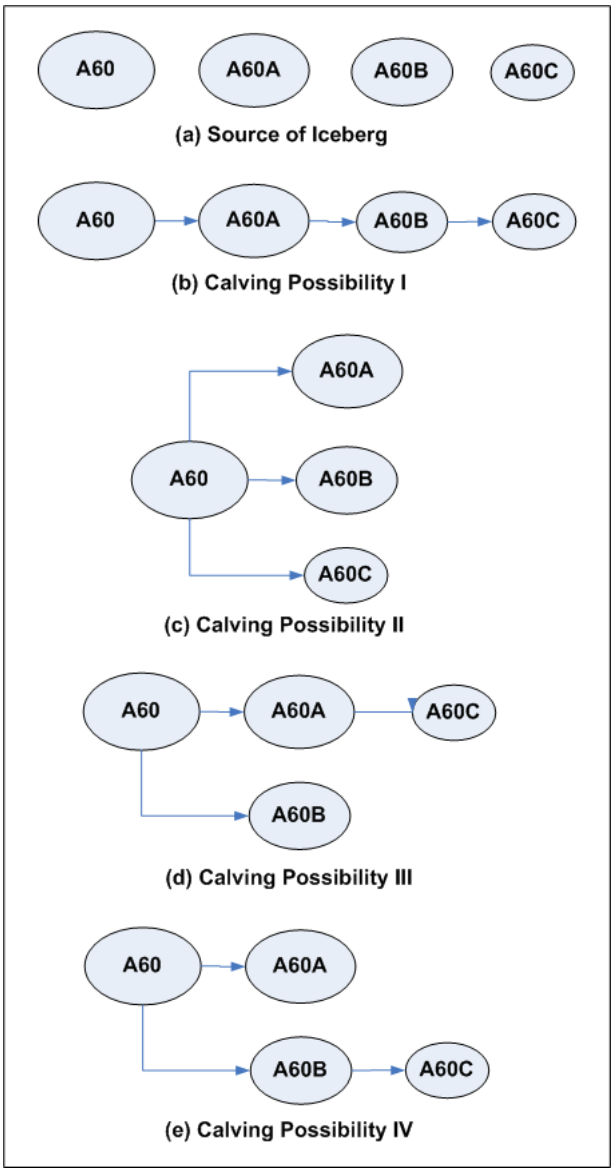

Figure 4.8: Calving possibilities that may occur to the group of iceberg A60

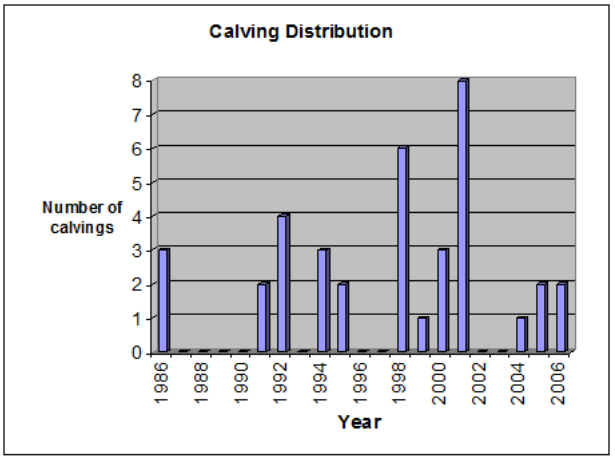

Figure 4.9: Calving distribution of icebergs in quadrant A of Antarctica

- A60C may be calved either from A60, A60A or A60B.

tr_detect_calve() is implemented as a function to handle the problem mentioned above. It works by comparing the first milestone of the child with the milestone of every candidate parent. Candidate parent is defined for iceberg(s) that has last suffix in its name smaller than the last suffix of the child's name. E.g. the candidate parent of A60B is A60 and A60A.

For detecting calving occurrence, there are three comparison parameters, i.e. size, time, and location. These parameters are used to define data constraints based on the following order:

1. Parent's size may not be smaller than child's size.
2. Calving is occurred on parent's milestone which has the shortest difference in time with the child's first milestone.
3. If there are more than one candidate parents that has the same difference in time with child, calving is defined on candidate parent that has the nearest location to the child.

Functions for detecting calving occurrence have been applied to 36 icebergs that were classified as child. The first calving event was identified in 1986 and the most recent ones is in 2006. Result of this detection can be used to analyze the number of calving distribution for each year as illustrated on figure 4.9. As can be seen on that figure, the most frequent calving was occurred in 2001 that brought out 8 new icebergs.

4.4 Back End Support for Trajectory Data Mining

Trajectory data mining has a goal to extract new knowledge from trajectory data set. Back end support for this purpose have been implemented by referring to the concept of relative motion pattern. The relative motion pattern has

Table 4.4: List of Function for Trajectory Data Mining

Name of Function	**Parameter Input**	**Return Type**
tr_remo_generate_interpolation	none	setof table
tr_remo_classification	none	setof table
tr_remo_assign_class_direction	direction_value: int	integer
tr_remo_assign_class_speed	speed_value: int	integer
tr_remo_constancy	searched_class: int, quantifier: int	setof table
tr_remo_concurrence	searched_class: int, searched_time: text	setof table
tr_remo_conformity	none	setof table

featured data mining approach, i.e. classification of parameters, to get similar pattern of object movements.

The list of implemented functions is shown on table 4.4.

This research uses speed and direction as parameter to detect similar patterns of iceberg movements. The steps of discovering similar pattern from iceberg data set is shown on figure 4.10. It starts by interpolating all iceberg in daily interval based on each iceberg's lifespan. The next step is calculating speed and direction for each interpolated value. Afterward, the value of speed and direction is classified into predefined class.

Two types of data classification have been applied. The first one is manual classification which is applied for classifying direction. This method is chosen with respect to cardinal point that commonly used for describing direction. The overall value of direction is distributed in the range of -180 degree to +180 degree. The second method is a quantile classification which is applied for classifying the speed value. This method is chosen to get an equal distribution of data set for each predefined number of class. Source code to classify speed and direction can be seen on appendix D.2.

To start the analysis of similarity pattern, the next step after data classification is the creation of relative motion matrix (see algorithm 7). This matrix is considered as a 2.5-dimensional analysis space with respect to the components that built the matrix [LI02, LIW05]:

- The column of matrix (horizontal axis) is termed as temporal axis since it has a sorted time steps. The range of time steps are defined by the user (refer to *time_begin* and *time_end*).
- The row of matrix (vertical axis) is defined as the object axis. The object axis is filled by icebergs that have movement in the specified time ranges.
- The matrix values is represented by the classified parameter.

After creation of remo matrix, the next step is finding the similar patterns. There are three types of pattern that can be identified.

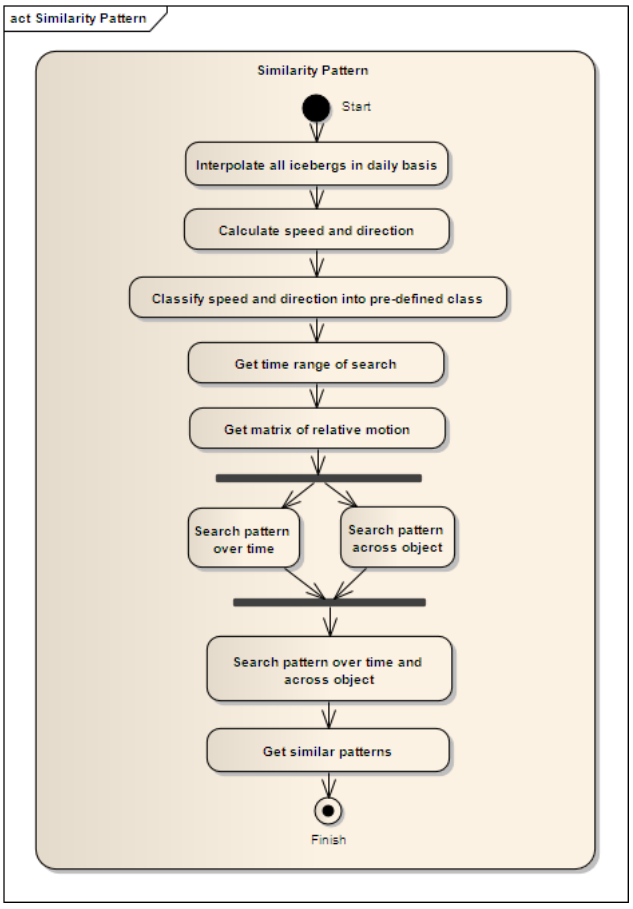

Figure 4.10: Activity diagram of trajectory data mining for searching similar pattern based on Relative Motion analysis

Algorithm 7: Create Relative Motion Matrix

Ensure: All icebergs have been interpolated in daily basis
Ensure: Table to store matrix ($remo\_matrix$) has been created
Ensure: Table reference has been ordered by date
Require: Time range ($time\_from, time\_to$)
1: **for all** iceberg name **do**
2: Take $first$ and $last$ milestone
3: **if** $first <= time\_from$ **and** $last >= time\_to$ **then**
4: retrieve all milestones which date fall in between the time range
5: store these milestones into table $remo\_matrix$
6: **end if**
7: **end for**

1. Similar pattern over horizontal axis which is termed as constancy.

   Function to detect constancy requires two parameters, i.e the intended class and the minimum length of the pattern (*search_class, quantifier*). This function works by scanning all matrix values in horizontal direction based on the searched class type. Pattern is identified for the sequence of value that have members equal or more than quantifier. Algorithm 8 shows how constancy is identified.

Algorithm 8: Search Template of Constancy

Ensure: Table reference has been ordered by iceberg name and date
Ensure: Table to store pattern of constancy ($remo\_constancy$) has been created
Require: Motion parameter and length of quantifier ($searched\_class, quantifier$)

```
Open cursor on table reference
Fetch one milestone (m_1)
loop
  if m_1.speed = searched_class or m_1.direction = searched_class then
    store milestone into array
    if array length >= quantifier then
      store array into table remo_constancy
      clean array
    end if
  end if
  Fetch one milestone as m_1
end loop
close cursor
```

2. Similar pattern across vertical axis which is termed as concurrence.

   Function to search concurrence also requires two parameters, i.e. the class type of motion parameter and the time search. It works by filtering the matrix based on predefined class and time search. The step to define concurrence is depicted on algorithm 9.

3. Similar pattern over horizontal and across vertical axis which is termed as conformity.

   Function to search conformity works by searching identical patterns in the result of constancy. The output of this function are pairs of iceberg that have similar pattern over time. Algorithm 10 shows how conformity is identified.

These functions above can be categorized as tools for deterministic searching since enable user to define the criteria of pattern that user needs. These functions have been applied to search similar pattern from 10 July 2000 to 10 August 2000 to icebergs in quadrant A. The identified patterns are as following:

1. Constancy are found on iceberg A22A, A38B, A39, A41, A43A and A43B which have similar direction of 45° for at least 5 consecutive days.

Algorithm 9: Search Template of Concurrence

Ensure: Table reference has been ordered by iceberg name and date
Ensure: Table to store pattern of concurrence ($remo\_concurrence$) has been created
Require: Motion parameter and time ($searched\_class, searched\_time$)
1: Open cursor on table reference where date = $search\_time$
2: Fetch one milestone (m_1)
3: **loop**
4: **if** $m_1.speed = searched\_class$ **or** $m_1.direction = searched\_class$ **then**
5: store milestone into table $remo\_concurrence$
6: **end if**
7: Fetch one milestone as m_1
8: **end loop**
9: close cursor

Algorithm 10: Search Template of Conformity

Ensure: Table reference has been ordered by iceberg name and date
Ensure: Table to store pattern of conformity ($remo\_conformity$) has been created
Ensure: Table to store temporary pattern of conformity ($remo\_conformity\_temp$) has been created
1: Open cursor on table reference
2: Fetch two milestones (m_1, m_2)
3: **loop**
4: **if** $m_1.name = m_2.name$ **and** $(m_2.date - m_1.date) = 1$ **then**
5: store milestone into array
6: **if** array length > 1 **then**
7: store array into table $remo\_conformity\_temp$
8: clean array
9: **end if**
10: **end if**
11: Fetch one milestone as m_1
12: **end loop**
13: close cursor
14: open cursor on table $remo\_conformity\_temp$
15: fetch two patterns (p_1, p_2)
16: **loop**
17: **if** $p_1 = p_2$ **then**
18: store p_1 and p_2 into table $remo\_conformity$
19: **end if**
20: $p_1 \leftarrow p_2$
21: fetch one pattern as p_2
22: **end loop**
23: close cursor

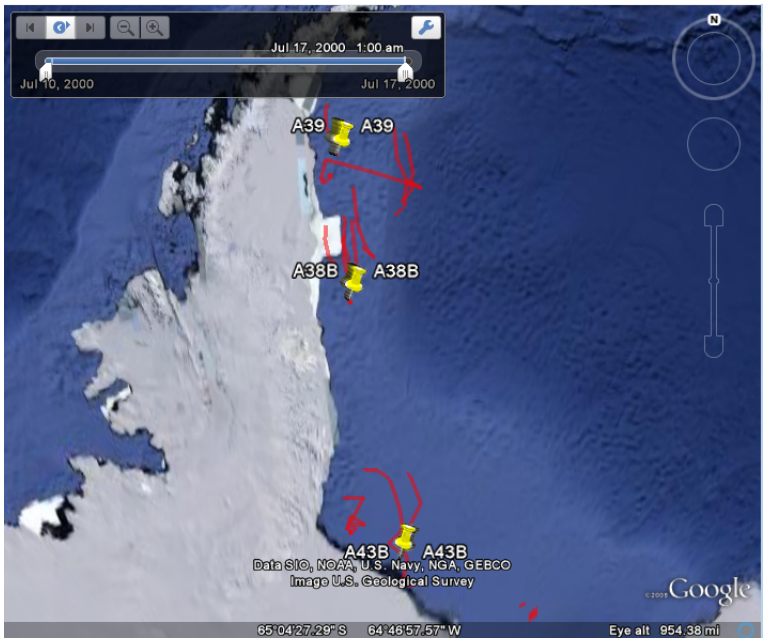

Figure 4.11: The Identified conformity on icebergs A38B, A39 and A43B from 10 to 17 July 2000

2. Concurrence are found on iceberg A22C, A35A, A35B, A35C, A38C, A39, A43B and A43C which have similar direction of 315° on 20 July 2000.

3. Conformity are found on iceberg A38B, A39 and A43B which have similar direction of 45° that occurred from 10 July 2000 to 17 July 2000 (see figure 4.11 and 4.12).

4.5 Summary

This chapter has presented the implementation of back end support for managing iceberg trajectory. Four types of functions dealt with data pre-processing, extraction of general information of trajectory, iceberg event detection and trajectory data mining. Data pre-processing involved functions for managing data from inconsistencies, empty values, duplicates and outliers. Extraction of general information of trajectory involved functions for data conversion, trajectory characteristics extraction, trajectory creation and trajectory summarization. Meanwhile regarding to event detection, two functions have been implemented for classifying general iceberg event and for detecting the position and the time of calving occurrence. The last functions were dealing with mining iceberg data set for searching similar patterns of iceberg movements. This research has implemented three types of deterministic pattern which are termed as constancy, concurrence and conformity.

Figure 4.12: The identified conformity with similar approximate direction of 45° in eight consecutive days

Chapter 5

Conclusions and Recommendations

Section 5.1 discusses conclusions that are drawn of applying back end support on icebergs trajectory and section 5.2 discusses recommendations for further development of trajectory back end support.

5.1 Conclusions

Conclusions are made by referring to the research objectives and questions that were specified on section 1.2. Two main results have been achieved: design of trajectory data model and implementation of back end support on icebergs trajectory.

5.1.1 Designing Conceptual Trajectory Data Model

Design of trajectory data model is based on trajectory definition in section 3.1.1. It is represented as UML class diagram (see section 3.2) which is related with the following research questions:

What data types, attributes, operations, classes and relationships are needed to represent general trajectories?

On chapter 3, trajectory has been defined as a collection of entities position sorted in time. Trajectory in DBMS is represented by using three main classes, i.e. moving entity, trajectory, milestone; and one association class termed as segment (see figure 3.4). The relationship among these classes shows that trajectory is part of a moving entity and each trajectory consists of at least two milestones. The term of milestone is referred to a record of moving entity that holds position and time as its main attributes. Data source which are obtained from position tracking devices, such as GPS or satellite radio collars, can be assumed as milestones.

The involved methods in class diagram are divided into three groups:

1. Methods for cleaning data from inconsistencies, empty values, duplicates and outliers. They are encapsulated in class Trajectory.

2. Methods for extracting general characteristics of trajectory. They are encapsulated in class Segment.

3. Method to construct or create trajectory as a linestring of point.

The required attributes to represent trajectory are listed as:

position as a point that is formed by latitude/longitude or XY Cartesian coordinates. It can be represented in degree unit or metric unit.

time of when position is recorded. It can be represented as date or integer data type.

milestone as a record of entity's movement that consists of position and time. It is represented as 3D geometry point.

trajectory that holds collection of milestones of one iceberg. It is represented as a linestring of geometry data type.

What functions are needed for extracting geometric properties from trajectories?

This research has defined four types of geometric properties (as defined in section 2.2.1), i.e. travelled time, travelled distance, speed and moving direction.

The list of functions to extract these properties is shown on table 4.2. They have been implemented as overloading methods which calculate either for the whole lifespan of trajectory or between two consecutive milestones (one segment).

Function to calculate travelled distance, direction and to create a trajectory were based on PostGIS functions, such as *st_distance()*, *st_azimuth()* and *st_makeline_garray()*. These functions require point data type as parameter inputs. There are three types of point that have been used:

1. Point in degree unit to calculate major direction of two points. To define this point, WGS84 Projection System and PostGIS function *st_makepoint(double precision, double precision)* were used.

2. Point in metric unit to calculate travelled distance between two milestones. To define this point, Antarctic Polar Projection System and PostGIS function *st_transtorm(geometry, integer)* were used.

3. Point in 3D geometry point to create a trajectory. The 3D geometry point applies on a milestone which consists of position and time. To define this point, WGS84 Projection System and PostGIS function *st_makepointm(double precision, double precision, integer)* were used.

Research has also implemented a function to summarize all these characteristics. From this summary, user may find minimum, maximum and average value of movement for all moving entities during their lifespan. Furthermore, summary information may also help users to define some constraints to the data set. E.g. for iceberg event detection, iceberg that has zero value in travelled distance is assumed as grounded icebergs. Summary information of trajectory characteristics has been illustrated on figure 4.6.

What functions are needed for extracting semantic properties from trajectories?

Semantic properties are dealing with extraction of new knowledge from trajectories. As has been discussed on section 2.2, semantic properties deal with the application requirements toward trajectory. Analysis of user requirements has to be done as preliminary step before extracting new knowledge from trajectory.

Data mining provides some techniques to extract the new knowledge from trajectory. In general, data mining involves three main functions:

1. Functions for data pre-processing: to transform the raw source data into an appropriate form for the subsequent analysis

2. Functions for data mining: to transform the prepared data into patterns or models.

3. Functions for post-processing of data mining results: to assess validity and usefulness of the extracted pattern and presents interesting knowledge to users by using appropriate visual tools.

5.1.2 Implementation of Trajectory Back end Supports on Iceberg Trajectory

Research has implemented back end support as database functions to manage iceberg trajectory in an open source DBMS of PostgreSQL. These functions are implemented by using programming language of PL/pgSQL and using some PostGIS functions. It is done based on the following research questions:

What are the user requirements on icebergs trajectory?

There are four requirements of icebergs trajectory (see section 2.1.2). These requirements are dealing with data pre-processing, the trajectory characteristics extraction, iceberg event detection and trajectory data mining to get similar pattern of iceberg movements.

These requirements have been illustrated as UML use case diagram (see figure 2.4) that involve some actors and use cases.

The relationship between actors and use cases are defined as following:

NIC has a role as data provider who records the movement of icebergs. NIC provides a raw data set that needs to be refined before it can be used in

further process. Many errors are found in the data set, such as data inconsistencies, empty values in some attributes, data duplicates and outliers. Hence to extract a reliable trajectory information, data pre-processing has to be included as a preliminary step.

Climatologist has necessity to detect event that occur on iceberg. Life history of iceberg is consider as one of invaluable factors for climate related investigations. Event detection, especially calving detection, is difficult to trace from the data set (see section 4.3). Hence, use case of event detection is provided which also includes use case of data pre-processing and trajectory data extraction.

Iceberg specialist has two necessities which are dealing with iceberg event detection and detection of similarly moving pattern. Use case of similarity pattern detection is required to reveal the behaviour of iceberg movement. Some users will get benefit from this analysis, such as navigator to define new shipping lanes and petroleum company to plant their oil pipeline in the safe area. Similar patterns can be detected by applying relative motion patterns analysis. This analysis includes data mining technique, i.e. classification of motion parameters, to create a semi-spatial temporal matrix as the basis to search patterns.

What is the required data model to represent icebergs trajectory?

Based on user requirement of iceberg movements (see section 2.1.2), icebergs trajectory have been modelled as UML class diagram (see figure 3.5). This model extends the general trajectory model by adding some new classes for detecting iceberg event and searching similarity pattern. These new classes have been illustrated in figure 3.6 and figure 3.7.

There are some required operations in iceberg trajectory data model which are listed as:

1. Methods for data conversion, i.e. to convert size from character into numerical type and to convert date from iceberg date into calender (Gregorian) format.

2. Methods to detect iceberg event, i.e. to classify general event and to detect the calving occurrence.

3. Methods to detect similar pattern, i.e. to detect pattern over times, across objects and combination of these two parameters, i.e. over times and across objects.

What steps are required for pre-processing data?

Steps in data pre-processing has been discussed in chapter 4 (see section 4.1) and the list of implemented functions can be seen on table 4.1. Data pre-processing is conducted based on four steps: data integration, data reduction, data cleaning and data transformation.

1. **Data integration** deals with integrating data source which is consists of four spread sheets file into one working space. Research has integrated 301 icebergs with 15737 records as one working table.

2. **Data reduction** deals with selecting only required attributes for data process. There are five attributes that have been chosen as basic information of iceberg trajectory, i.e. iceberg name, longitude, latitude, date and size.

3. **Data cleaning** deals with cleaning data set from errors. There are four types of errors have been identified.

   (a) Data inconsistencies found in the name of the iceberg and latitude values. Some icebergs name have mixed upper and lowercase letters. There are also typing errors in the latitude, where they should have a negative value. In accordance with the requirements of the NIC (see section 2.1.1), iceberg can not be located on the north of the equator.

   (b) Empty values which are found either in latitude, longitude, size or in both position and date attributes.

   (c) Data duplicates are found as full duplicates (i.e. identical value in iceberg name, date, latitude and longitude) and partial duplicates (i.e. identical value in iceberg name and date).

   (d) Outliers which are found on iceberg movements that have speed more than 80 km/day.

4. **Data transformation** deals with transforming some attributes into required data types. These attributes are size, date, latitude and longitude.

What functions are used to detect iceberg events, such as calving, grounded and floating; and also parent child relationship among icebergs?

The required functions to detect iceberg event has been discussed in chapter 4 section 4.3. Data model of event detection has also been illustrated in figure 3.6. These functions have been encapsulated into two classes: *Iceberg_Event* and *Calving_Iceberg*. There are two main functions are involved:

- Function to classify all icebergs into one of general event, i.e. calving, grounded or floating.

  Event classification is based on iceberg name and travelled distance. Iceberg that has similar main name to other icebergs is classified as calving and for one single name is classified as non-calving. Afterward, the travelled distance of non-calving icebergs is calculated. For those which have zero in travelled distance are classified as grounded, otherwise they assumed to be floating.

- Function to detect calving occurrence, i.e. to define from which iceberg it was calved and also the position and time of calving.

Calving detection is required with respect to the way of NIC define iceberg name (see section 2.1.1) which creates difficulty to trace the parent child relationship between icebergs (see figure 4.8).

Function to detect the occurrence of calving is implemented based on the constraints that are defined on section 4.3. Calving is assumed to occur on a milestone of parent's trajectory. There are three parameters are considered, i.e. size, time and position. The following constraints are defined that child's size must be smaller than the parent and child's age must also be younger than the parent's. An ideal calving position is defined as a nearest position between the first milestone of a child to its parent's trajectory.

What functions are used to detect similar movement patterns of iceberg?

Similarly movement patterns are extracted based on the concept of relative motion patterns. As has been discussed on section 4.4, the needed functions are related with:

1. Interpolate all icebergs based on their lifespan to daily intervals.
2. Classify the motion parameters, i.e. speed and direction, into a number of predefined classes.
3. Create a matrix as the basis for searching the patterns.
4. Search patterns which parameters are defined by the user.

These functions above can be categorized as tools for deterministic searching since users may define the criteria of patterns that users look for. The list of functions that have been implemented can be seen on table 4.4.

There are three types of pattern that can be identified as:

- Constancy for similar pattern over times which enable users to define the minimum length of pattern and the type of parameter.
- Concurrence for similar pattern across some icebergs which enable users to specify the time and the type of parameter.
- Conformity for similar pattern over times and across some icebergs.

5.2 Recommendations

Based on implementation of back end support for managing and extracting some information from icebergs trajectory, the following recommendations are made for future improvement:

1. Provide back end support to extract more trajectory characteristics.

   This research has focus on characteristics of individual trajectory. Meanwhile, trajectory may also be characterized by their relationship to other trajectories, such as place of intersection, distance between two trajectories or order of duration among trajectories. Back end support to extract this additional characteristic need to be implemented since it enables users to cluster trajectories based on their spatial and temporal proximity.

2. Develop icebergs trajectory for other user requirements.

   Icebergs trajectory deal with various type of users, such as oceanographers, glaciologists, climatologists, oil companies and navigators. They might require iceberg information that have not been covered in this research, such as the evolution of iceberg distribution, the influence of external phenomena on iceberg movements and the influence of external factor to calving site and size reduction.

3. Improve the classification technique by using statistical computing language, such as PL/R that can be integrated with functions from PostgreSQL.

   One requirement of trajectory data mining is ability to provide function for statistical analysis, such as data classification or clustering. Integration of this required function to database will make system to have more abilities to explore new knowledge from trajectory data sets. PL/R is a recommended statistical package to be installed in PostgreSQL database system since it has features on statistics matters which can be developed by using similar declaration of PostgreSQL functions.

4. Enhance prototype by adding various motion parameters and search template for patterns.

   The three patterns that have been applied in this research are categorized as patterns without neighbourhood information. Further research on trajectory similarity needs to deal with searching pattern that involve information from trajectory environment. By doing this, the location of individual trajectory can be measured relatively to other trajectories. Some patterns have been identified for this purpose, such as flock, convergence and divergence. According to Laube [LIW05], flock can be used to detect the center of clusters, convergence can be used to detect a group of objects that simultaneously heading for a specified position and divergence can be used to describe a group that disperse.

Bibliography

[AAPS08] Natalia Andrienko, Gennady Andrienko, Nikos Pelekis, and Stefano Spaccapietra. *Mobility, Data Mining and Privacy*, chapter Basic concept of movement data, pages 15–38. Springer, 2008.

[ABdM+07] Luis O. Alvares, Vania Bogorny, Jose A. de Macedo, Bart Moelans, and Stefano Spaccapietra. Dynamic modeling of trajectory patterns using data mining and reverse engineering. In *ER '07: Tutorials, posters, panels and industrial contributions at the 26th international conference on Conceptual modeling*, pages 149–154. Australian Computer Society, Inc., 2007.

[BL02] Jarom Ballantyne and David G. Long. A multidecadal study of the number of antarctic icebergs using scatterometer data. In *IEEE International Geoscience and Remote Sensing Symposium (IGARSS 2002), and 24th Canadian Symposium on Remote Sensing*, volume V, pages 3029–3031. IEEE International, IEEE Computer Society, 2002.

[BPT04] Sotiris Brakatsoulas, Dieter Pfoser, and Nectaria Tryfona. Modeling, storing, and mining moving object databases. In *IDEAS '04: Proceedings of the International Database Engineering and Applications Symposium*, pages 68–77. IEEE Computer Society, 2004.

[Bri] Encyclopedia Britannica. Origin of icebergs. http://www.britannica.com/EBchecked/topic/281212/iceberg accessed on November 2009.

[BTKP09] Connie Blok, Ulanbek Turdukulov, Barend Köbben, and Juan L. Pomares. Web-based visual exploration and error detection in large data sets: antarctic iceberg tracking data as a case. In *Proceedings of the 24th international cartographic conference ICC : The world's geo - spatial solutions*, pages 10–pp. International Cartographic Association (ICA), 2009.

[BWM07] Douglas I. Benn, Charles R. Warren, and Ruth H. Mottram. Calving processes and the dynamics of calving glaciers. *Earth-Science Reviews*, 82(3-4):143–179, 2007.

[Cen] National Ice Center. Antarctica icebergs. http://www.natice.noaa.gov/products/iceberg accessed on October 2009.

[FPSS96] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17:37–54, 1996.

[GBE+00] Ralf H. Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.*, 25:1–42, 2000.

[Gro09] PostgreSQL Global Development Group. Postgresql 8.4.2 documentation. Technical report, The PostgreSQL Global Development Group, Available at http://www.postgresql.org/docs/8.4/static/release-8-4.html, 2009.

[GS05] Ralf H. Güting and Markus Schneider. *Moving object databases*. Morgan Kaufmann, 2005.

[HK06] Jiawei Han and Micheline Kamber. *Data mining concept and technique*. Morgan Kaufmann, 2006.

[LI02] Patrick Laube and Stephan Imfeld. Analyzing relative motion within rroups of trackable moving point objects. In *Lecture Notes in Computer Science*, pages 132–144. Springer Berlin / Heidelberg, 2002.

[LIW05] Patrick Laube, Stephan Imfeld, and Robert Weibel. Discovering relative motion patterns in groups of moving point objects. In *International Journal of Geographical Information Science*, volume 19, pages 639– 68. Taylor and Francis Ltd, 2005.

[LvKI05] Patrick Laube, Marc van Kreveld, and Stephan Imfeld. Finding remo — detecting relative motion patterns in geospatial lifelines. In *Developments in Spatial Data Handling*, pages 201–215. Springer Berlin Heidelberg, 2005.

[Mar80] Fred Maryanski. Backend database system. *ACM Computing Surveys*, 12(1):3–25, 1980.

[MH09] Harvey J. Miller and Jiawei Han. *Geographic data mining and knowledge discovery (2nd edition)*. Chapman and Hall Book, 2009.

[MS04] Hoda M. Mokhtar and Jianwen Su. Universal trajectory queries for moving object databases. In *Mobile Data Management, IEEE International Conference*, pages 12–pp. IEEE Computer Society, 2004.

[NRRT05] Mirco Nanni, Alessandra Raffaetà, Chiara Renso, and Franco Turini. *Applications of declarative programming and knowledge management*, volume 3392/2005, chapter Deductive and Inductive Reasoning on Spatio-Temporal Data, pages 98–115. Springer Berlin / Heidelberg, 2005.

[PKM+07] Nikos Pelekis, Ioannis Kopanakis, Gerasimos Marketos, Irene Ntoutsi, Gennady Andrienko, and Yannis Theodoridis. Similarity search in trajectory databases. In *Proceedings of the 14th International Symposium on Temporal Representation and Reasoning*, pages 129–140. IEEE Computer Society, 2007.

[Pro09] PostGIS Refractions Research Project. Postgis 1.4.0 manual. Technical report, PostGIS Refractions Research Project, available at http://postgis.refractions.net/documentation/manual-1.4/, 2009.

[Ra80] R. Q. Robe and D. C. M aier. Long-term drift of icebergs in baffin bay and the labrador sea. In *Cold Regions Science and Technology*, volume 1 of *3-4*, pages 183–193, 1980.

[SC09] National Snow and Ice Data Center. Quick facts on icebergs. http://nsidc.org/quickfacts/icebergs.html accessed on November 2009, 2009.

[SPD+08] Stefano Spaccapietra, Christine Parent, Maria L. Damiani, Jose A. de Macedo, Fabio Porto, and Christelle Vangenot. A conceptual view on trajectories. In *Data Knowledge Engineering*, volume 65, pages 126–146. Elsevier Science Publishers B. V., 2008.

[SYW02] Andreas Schmittner, Masakazu Yoshimori, and Andrew J. Weaver. Instability of glacial climate in a model of the ocean-atmosphere-cryosphere system. *Science Express*, 295(5559):8–pp, 2002.

[TJ99] Nectaria Tryfona and Christian S. Jensen. Conceptual data modeling for spatiotemporal applications. *GeoInformatica*, 3:245–268, 1999.

[TPN+09] E. Tiakas, A.N. Papadopoulos, A. Nanopoulos, Y. Manolopoulos, Dragan Stojanovic, and Slobodanka Djordjevic-Kajan. Searching for similar trajectories in spatial networks. In *Journal of System Software*, volume 82, pages 772–788. Elsevier Science Inc., 2009.

[VD00] Brian Veitch and Claude Daley. Iceberg evolution modeling a background study. Technical report, Faculty of Engineering and Applied Science Memorial University of Newfoundland, 2000.

[WXCJ98] Ouri Wolfson, Bo Xu, Sam Chamberlain, and Liqin Jiang. Moving objects databases: issues and solutions. In *Proceedings of the 10th international Conference on Scientific and Statistical database Management*, pages 111–122. IEEE Computer Society, 1998.

Appendix A

Back end Support for Data Pre-processing

A.1 Data Integration

Listing A.1: Create Table temp1

```
CREATE TABLE iceberg_temp1(
        gid serial primary key not null,
        iceberg character varying,
        date integer,
        latitude double precision,
        longitude double precision,
        size character varying
);
```

Listing A.2: Data Integration

```
CREATE OR REPLACE FUNCTION tr_data_integration() RETURNS
   TEXT AS $$
DECLARE
        var_a quad_a%rowtype;
        var_b quad_b%rowtype;
  var_c quad_c%rowtype;
  var_d quad_d%rowtype;
BEGIN
for var_a in select * from quad_a
loop
        insert into iceberg_temp1(iceberg, date, latitude,
           longitude, size)
        values (var_a.iceberg, var_a.date, var_a.latitude,
           var_a.longitude, var_a.size);
end loop;
for var_b in select * from quad_b
loop
```

```
        insert into iceberg_temp1(iceberg, date, latitude,
           longitude, size)
        values (var_b.iceberg, var_b.date, var_b.latitude,
           var_b.longitude, var_b.size);
end loop;
for var_c in select * from quad_c
loop
        insert into iceberg_temp1(iceberg, date, latitude,
           longitude, size)
        values (var_c.iceberg, var_c.date, var_c.latitude,
           var_c.longitude, var_c.size);
end loop;
for var_d in select * from quad_d
loop
        insert into iceberg_temp1(iceberg, date, latitude,
           longitude, size)
        values (var_d.iceberg, var_d.date, var_d.latitude,
           var_d.longitude, var_d.size);
end loop;
execute 'update iceberg_temp1 set iceberg = upper(iceberg)';
return ('check table iceberg_temp1');
END;
$$ LANGUAGE 'plpgsql';
```

A.2 Data Consistency Management

A.3 Empty Value Management

Listing A.3: Manage Empty Value in Position Time

```
CREATE OR REPLACE FUNCTION tr_manage_empty_position_time()
   RETURNS SETOF iceberg_temp1 AS $$
DECLARE
    nullValue iceberg_temp1%rowtype;
BEGIN
for nullValue in select * from iceberg_temp1
loop
        delete from iceberg_temp1 where (date is null) and (
           latitude is null or latitude = 0) and (longitude
           is null or longitude = 0);
    return next nullValue;
end loop;
return;
END;
$$ LANGUAGE 'plpgsql';
```

Listing A.4: Manage Empty Size

```
CREATE OR REPLACE FUNCTION tr_manage_empty_size() RETURNS
   SETOF iceberg_temp1 AS $$
DECLARE
    cur refcursor;
    var1 iceberg_temp1%rowtype;
    var2 iceberg_temp1%rowtype;
BEGIN
open cur for select * from iceberg_temp1 order by iceberg,
   date;
fetch cur into var1;
fetch cur into var2;
loop
    if (var1.iceberg = var2.iceberg) and (var2.size is null)
        then
        update iceberg_temp1 set size = var1.size where
           iceberg = var2.iceberg;
                var2.size = var1.size;
        else
        if (var1.iceberg = var2.iceberg) and (var1.size is
           null) then
            delete from iceberg_temp1 where iceberg = var1.
               iceberg;
        end if;
        end if;
    var1 = var2;
        fetch cur into var2;
        exit when not found;
end loop;
close cur;
END;
$$ LANGUAGE 'plpgsql';
```

Listing A.5: Manage Empty Latitude

```
CREATE OR REPLACE FUNCTION tr_manage_empty_latitude()
   RETURNS SETOF iceberg_temp1 AS $$
DECLARE
    cur refcursor;
    var1 iceberg_temp1%rowtype;
    var2 iceberg_temp1%rowtype;
    var3 iceberg_temp1%rowtype;
    new_latitude double precision;
BEGIN
open cur for select * from iceberg_temp1 order by iceberg,
   date;
```

```
fetch cur into var1;
fetch cur into var2;
fetch cur into var3;
loop
    if (var1.iceberg = var2.iceberg) and (var2.iceberg =
       var3.iceberg) and (var2.latitude = 0) then
        new_latitude = ((var3.latitude - var1.latitude) /
           tr_time_difference(var3.date, var1.date)) + var1.
           latitude;
        update iceberg_temp1 set latitude = new_latitude
           where gid = var2.gid;
    end if;
    var1 = var2;
    var2 = var3;
    fetch cur into var3;
    exit when not found;
end loop;
close cur;
END;
$$ LANGUAGE 'plpgsql';
```

Listing A.6: Manage Empty Longitude

```
CREATE OR REPLACE FUNCTION tr_manage_empty_longitude()
   RETURNS SETOF iceberg_temp1 AS $$
DECLARE
    cur refcursor;
    var1 iceberg_temp1%rowtype;
    var2 iceberg_temp1%rowtype;
    var3 iceberg_temp1%rowtype;
    new_longitude double precision;
BEGIN
open cur for select * from iceberg_temp1 order by iceberg,
   date;
fetch cur into var1;
fetch cur into var2;
fetch cur into var3;
loop
    if (var1.iceberg = var2.iceberg) and (var2.iceberg =
       var3.iceberg) and (var2.longitude = 0) then
        new_longitude = ((var3.longitude - var1.longitude) /
            tr_time_difference(var3.date, var1.date)) + var1
           .longitude;
        update iceberg_temp1 set longitude = new_longitude
           where gid = var2.gid;
    end if;
    var1 = var2;
```

```
    var2 = var3;
    fetch cur into var3;
    exit when not found;
end loop;
if (var1.iceberg = var2.iceberg) and (var2.longitude = 0)
   then
    update iceberg_temp1 set longitude = var1.longitude
       where gid = var2.gid;
end if;
close cur;
end;
$$ LANGUAGE 'plpgsql';
```

A.4 Data Duplicate Management

Listing A.7: Create Table temp2

```
CREATE TABLE iceberg_temp2 (
    iceberg character varying,
    date integer,
    latitude double precision,
    longitude double precision,
    size character varying
);
```

Listing A.8: Manage Duplicate Full

```
CREATE OR REPLACE FUNCTION tr_manage_duplicate_full()
   RETURNS SETOF refcursor AS $$
DECLARE
    cur refcursor;
    var1 iceberg_temp1%rowtype;
    var2 iceberg_temp1%rowtype;
BEGIN
open cur for (select * from iceberg_temp1 order by iceberg,
   date);
fetch cur into var1;
fetch cur into var2;
if not((var1.iceberg = var2.iceberg) and (var1.date = var2.
   date) and (var1.latitude = var2.latitude) and (var1.
   longitude = var2.longitude)) then
    insert into iceberg_temp2 values (var1.iceberg, var1.
       date, var1.latitude, var1.longitude, var1.size);
end if;
loop
var1 = var2;
```

```
fetch cur into var2;
exit when not found;
    if not((var1.iceberg = var2.iceberg) and (var1.date =
       var2.date) and (var1.latitude = var2.latitude) and (
       var1.longitude = var2.longitude)) then
        insert into iceberg_temp2 values (var1.iceberg, var1
           .date, var1.latitude, var1.longitude, var1.size);
    end if;
end loop;
close cur;
END;
$$ LANGUAGE 'plpgsql'
```

Listing A.9: Create Table temp3

```
CREATE TABLE iceberg_temp3(
    iceberg character varying,
    date integer,
    latitude double precision,
    longitude double precision,
    size character varying
);
```

Listing A.10: Manage Duplicate Partial

```
CREATE OR REPLACE FUNCTION tr_manage_duplicate_partial()
   RETURNS SETOF refcursor AS $$
DECLARE
    data iceberg_temp2%rowtype;
BEGIN
for data in SELECT * FROM iceberg_temp2 WHERE (iceberg, date
   ) IN (select iceberg, date from iceberg_temp2 group by
   iceberg, date having count(*) = 1) order by iceberg, date
loop
    insert into iceberg_temp3 values (data.iceberg, data.
       date, data.latitude, data.longitude, data.size);
end loop;
for data in select iceberg, date, avg(latitude), avg(
   longitude) from iceberg_temp2 where (iceberg, date) in (
   select iceberg, date from iceberg_temp3 group by iceberg,
    date having count(*) > 1) group by iceberg, date order
   by iceberg, date
loop
    insert into iceberg_temp3 values (data.iceberg, data.
       date, data.latitude, data.longitude, data.size);
end loop;
return;
```

```
END;
$$ LANGUAGE 'plpgsql';
```

Listing A.11: Update Geometry Column

```
select addgeometrycolumn('public', 'iceberg_temp3', '
   geom_degree', 4326, 'POINT', 2);
update iceberg_temp3 set geom_degree = setsrid(makepoint(
   longitude, latitude),4326);

select addgeometrycolumn('public', 'iceberg_temp3', '
   geom_metric', 3031, 'POINT', 2);
update iceberg_temp3 set geom_metric = Transform(geom_degree
   , 3031);
```

A.5 Outliers Management

Listing A.12: Create Table Iceberg Free From Outliers

```
CREATE TABLE iceberg_clean(
        gid serial PRIMARY KEY NOT NULL,
    iceberg character varying,
    date integer,
    latitude double precision,
    longitude double precision,
    size character varying,
    pointm geometry
);
```

Listing A.13: Manage Outliers

```
CREATE OR REPLACE FUNCTION tr_manage_outlier() RETURNS SETOF
    refcursor AS $$
DECLARE
        cur refcursor;
        data iceberg_temp3%rowtype;
        var1 iceberg_temp3%rowtype;
        var2 iceberg_temp3%rowtype;
        var3 iceberg_temp3%rowtype;
        speed1 double precision;
    speed2 double precision;
        temp_longitude double precision;
    my_val integer;
        temp_geom_metric geometry;
    temp_geom_degree geometry;
        time_differ1 integer;
```

```
    time_differ2 integer;
        distance1 double precision;
    distance2 double precision;
        temp_distance1 double precision;
    temp_distance2 double precision;
        temp_speed1 double precision;
    temp_speed2 double precision;
BEGIN
for data in select distinct iceberg from iceberg_temp3
loop
open cur for select * from iceberg_temp3 where iceberg =
   data.iceberg order by iceberg, date ;
fetch cur into var1;
fetch cur into var2;
fetch cur into var3;
if (tr_num_milestone(data.iceberg) >= 3) then
insert into iceberg_clean (iceberg, date, latitude,
   longitude, size, pointm) values (var1.iceberg, var1.date,
    var1.latitude, var1.longitude, var1.size, ST_MakePointM(
   var1.longitude, var1.latitude, var1.date));
loop
if (var1.iceberg = var2.iceberg) and (var2.iceberg = var3.
   iceberg) then
    speed1 = tr_average_speed(var1.date, var1.geom_metric,
       var2.date, var2.geom_metric);
        if not(speed1 >= 80) then
                insert into iceberg_clean (iceberg, date,
                   latitude, longitude, size, pointm) values
                    (var2.iceberg, var2.date, var2.latitude,
                    var2.longitude, var2.size, ST_MakePointM
                   (var2.longitude, var2.latitude, var2.date
                   ));
        else
                if (abs(var1.longitude - var2.longitude) >=
                   10) then
                        temp_longitude = var2.longitude /
                           10;
                elsif (round(abs(var2.longitude / var1.
                   longitude)) = 1) then
                        if ((var1.longitude > 0) and (var2.
                           longitude < 0)) or ((var1.
                           longitude < 0) and (var2.
                           longitude > 0)) then
                       temp_longitude = var2.longitude * -1;
                        end if;
        else
                  temp_longitude = var2.longitude;
```

```
                end if;
                temp_geom_degree = setsrid(makepoint(
                   temp_longitude, var1.latitude),4326);
                temp_geom_metric = Transform(
                   temp_geom_degree,3031);
                temp_speed1 = tr_average_speed(var1.date,
                   var1.geom_metric, var2.date,
                   temp_geom_metric);
                if not(temp_speed1 >= 80) then
                      insert into iceberg_clean (iceberg,
                         date, latitude, longitude, size,
                         pointm) values (var2.iceberg,
                         var2.date, var2.latitude,
                         temp_longitude, var2.size,
                         ST_MakePointM(temp_longitude,
                         var2.latitude, var2.date));
                end if;
        end if;
end if;
var1 = var2;
var2 = var3;
fetch cur into var3;
exit when not found;
end loop;
speed1 = tr_average_speed(var1.date, var1.geom_metric, var2.
   date, var2.geom_metric);
if not(speed1 >= 80) then
        insert into iceberg_clean (iceberg, date, latitude,
           longitude, size, pointm) values (var2.iceberg,
           var2.date, var2.latitude, var2.longitude, var2.
           size, ST_MakePointM(var2.longitude, var2.latitude
           , var2.date));
end if;
elsif (tr_num_milestone(data.iceberg) = 2) then
        insert into iceberg_clean (iceberg, date, latitude,
           longitude, size, pointm) values (var1.iceberg,
           var1.date, var1.latitude, var1.longitude, var1.
           size, ST_MakePointM(var1.longitude, var1.latitude
           , var1.date));
        insert into iceberg_clean (iceberg, date, latitude,
           longitude, size, pointm) values (var2.iceberg,
           var2.date, var2.latitude, var2.longitude, var2.
           size, ST_MakePointM(var2.longitude, var2.latitude
           , var2.date));
else
        insert into iceberg_clean (iceberg, date, latitude,
           longitude, size, pointm) values (var1.iceberg,
```

```
            var1.date, var1.latitude, var1.longitude, var1.
            size, ST_MakePointM(var1.longitude, var1.latitude
            , var1.date));
end if;
close cur;
end loop;
END;
$$ LANGUAGE 'plpgsql';
```

Appendix B

Back end Support for Trajectory Data Extraction

B.1 Data Conversion

Listing B.1: Convert Date To Gregorian

```
CREATE OR REPLACE FUNCTION tr_julian_to_calender(julian_date
    integer) RETURNS text AS $$
DECLARE
    day integer; month integer; year integer; num integer[];
    day_text text; month_text text; year_text text;
BEGIN
year = cast(substring(cast(julian_date as text) from 1 for
   4) as int);
day = cast(substring(cast(julian_date as text) from 5 for 3)
    as int);
if tr_is_leap_year(year) = true then
    num = array[31,29,31,30,31,30,31,31,30,31,30,31];
else
    num = array[31,28,31,30,31,30,31,31,30,31,30,31];
end if;
if day <= 31 then
    month = 1;
else
    for i in 1..(array_upper(num,1)+1)
    loop
        if (day - num[i]) > num[i+1] then
            day = day - num[i];
            continue;
        else
            day = day - num[i];
            month = i+1;
            exit;
```

```
        end if;
    end loop;
end if;
if day < 10 then
    day_text = cast(day as text);
    day_text = '0' || day_text;
else
    day_text = cast(day as text);
end if;
if month < 10 then
    month_text = cast(month as text);
    month_text = '0' || month_text;
else
    month_text = cast(month as text);
end if;
year_text = cast(year as text);
return (day_text || '-' || month_text || '-' || year_text);
END;
$$ LANGUAGE 'plpgsql';
```

Listing B.2: Convert Gregorian To Iceberg Date

```
CREATE OR REPLACE FUNCTION tr_calender_to_julian(calender
   character varying) RETURNS integer AS $$
DECLARE
    day integer; month integer; year integer;
    temp_day text; temp_year text; temp_date text;
       final_date integer;
BEGIN
day = cast(substring(calender from 1 for 2) as int);
month = cast(substring(calender from 4 for 2) as int);
year = cast(substring(calender from 7 for 4) as int);
if month = 1 then
    day = day;
elsif month = 2 then
    day = day + 31;
elsif month = 3 then
    day = day + 59;
elsif month = 4 then
    day = day + 90;
elsif month = 5 then
    day = day + 120;
elsif month = 6 then
    day = day + 151;
elsif month = 7 then
    day = day + 181;
elsif month = 8 then
```

```
    day = day + 212;
elsif month = 9 then
    day = day + 242;
elsif month = 10 then
    day = day + 273;
elsif month = 11 then
    day = day + 303;
else
    day = day + 334;
end if;
if (month >= 3) and (tr_is_leap_year(year) = true) then
    day = day + 1;
end if;
if day < 100 then
    temp_day = cast(day as text);
    temp_day = '0' || temp_day;
else
    temp_day = cast(day as text);
end if;
temp_year = cast(year as text);
temp_date = temp_year || temp_day;
final_date = cast(temp_date as int);
return final_date;
END;
$$ LANGUAGE 'plpgsql';
```

Listing B.3: Calculate Size in NM

```
CREATE OR REPLACE FUNCTION tr_calculate_size_in_nm(character
    varying, character varying) RETURNS text AS $$
DECLARE
    table_name alias for $1;
    column_size_name alias for $2;
BEGIN
execute 'alter table ' || quote_ident(table_name) || ' add
   column ' || quote_ident(column_size_name) || ' integer';
execute 'update ' || quote_ident (table_name) || ' set ' ||
   quote_ident(column_size_name) || ' = cast(substring(size
   from 1 for 2) as int) * cast(substring(size from 4 for 2)
    as int)';
return table_name;
END;
$$ LANGUAGE 'plpgsql';
```

Listing B.4: Calculate Size in KM

```
CREATE OR REPLACE FUNCTION tr_calculate_size_in_km(character
    varying, character varying)
  RETURNS text AS $$
DECLARE
    table_name alias for $1;
column_size_name alias for $2;
BEGIN
execute 'alter table ' || quote_ident(table_name) || ' add
   column ' || quote_ident(column_size_name) || ' double
   precision';
execute 'update ' || quote_ident (table_name) || ' set ' ||
   quote_ident(column_size_name) || ' = (cast(substring(size
    from 1 for 2) as int) * cast(substring(size from 4 for
   2) as int) * 3.4343900)';
return table_name;
END;
$$ LANGUAGE 'plpgsql';
```

B.2 Trajectory Characteristic Extraction

Listing B.5: Check Leap Year

```
CREATE OR REPLACE FUNCTION tr_is_leap_year(year1 integer)
   RETURNS boolean AS $$
DECLARE
BEGIN
if (year1 % 4 != 0) then
    return false;
else
    if year1 % 100 != 0 then
        return true;
    else
        if year1 % 400 != 0 then
            return false;
        else
            return true;
        end if;
    end if;
end if;
END;
$$ LANGUAGE 'plpgsql';
```

Listing B.6: Calculate Time Difference

```
CREATE OR REPLACE FUNCTION tr_time_difference(var1 integer,
   var2 integer) RETURNS integer AS $$
```

```
DECLARE
    diff integer;       diff_year integer;
    day1 integer;       day2 integer;
    year1 integer;      year2 integer;
BEGIN
year1 = cast(substring(cast(var1 as text) from 1 for 4) as
   int);
year2 = cast(substring(cast(var2 as text) from 1 for 4) as
   int);
day1 = cast(substring(cast(var1 as text) from 5 for 3) as
   int);
day2 = cast(substring(cast(var2 as text) from 5 for 3) as
   int);
diff_year = (year2 - year1);
diff = var2 - var1;
if tr_is_leap_year(year1) = true then
    if diff < 366 then
        return diff;
    else
        if diff_year = 1 then
            diff = (366 + day2) - day1;
        else
            diff = (diff_year * 366) + day2 - day1;
        end if;
        return diff;
    end if;
else
    if diff < 365 then
        return diff;
    else
        if diff_year = 1 then
               diff = (365 + day2) - day1;
        else
            diff = (diff_year * 365) + day2 - day1;
        end if;
        return diff;
    end if;
end if;
END;
$$ LANGUAGE 'plpgsql';
```

Listing B.7: Calculate Travelled Time

```
CREATE OR REPLACE FUNCTION tr_travelled_time(ice_id
   character varying) RETURNS INTEGER AS $$
DECLARE
    cur refcursor;
```

```
    var1 iceberg_clean%rowtype;
    var2 iceberg_clean%rowtype;
    diff integer;
    ave_time integer;
BEGIN
open cur for select * from iceberg_clean where iceberg =
   ice_id order by iceberg, date;
fetch cur into var1;
fetch cur into var2;
ave_time := 0;
loop
    diff = tr_time_difference(var1.date, var2.date);
    ave_time = ave_time + diff;
    var1 = var2;
    fetch cur into var2;
    exit when not found;
end loop;
return ave_time;
close cur;
END;
$$ LANGUAGE 'plpgsql';
```

Listing B.8: Calculate Travelled Distance

```
CREATE OR REPLACE FUNCTION tr_traveled_distance(ice_id
   character varying) RETURNS double precision AS $$
DECLARE
    cur refcursor;
    var1 geometry;
    var2 geometry;
    ave_dist double precision;
    diff double precision;
BEGIN
open cur for select geom_metric from table_iceberg3 where
   iceberg = ice_id order by iceberg, date;
fetch cur into var1;
fetch cur into var2;
ave_dist := 0.0;
loop
    diff = st_distance(var1,var2);
    ave_dist = ave_dist + diff;
    var1 = var2;
    fetch cur into var2;
    exit when not found;
end loop;
close cur;
return ave_dist/1000;
```

```
END;
$$ LANGUAGE 'plpgsql';
```

Listing B.9: Calculate Speed 1

```
CREATE OR REPLACE FUNCTION tr_speed(ice_id character varying
    ) RETURNS double precision AS $$
DECLARE
     ave_time integer;
     ave_distance double precision;
BEGIN
ave_time = tr_traveled_time(ice_id);
ave_distance = tr_traveled_distance(ice_id);
return ave_distance/ave_time;
END;
$$ LANGUAGE 'plpgsql';
```

Listing B.10: Calculate Speed 2

```
CREATE OR REPLACE FUNCTION tr_speed(date1 integer,
    geom_metric1 geometry, date2 integer, geom_metric2
    geometry) RETURNS double precision AS $$
DECLARE
     ave_time integer;
     ave_distance double precision;
BEGIN
ave_time = tr_time_difference(date1,date2);
ave_distance = st_distance(geom_metric1, geom_metric2)/1000;
return ave_distance/ave_time;
END;
$$ LANGUAGE 'plpgsql';
```

Listing B.11: Calculate Direction 1

```
CREATE OR REPLACE FUNCTION tr_direction(ice_id character
    varying) RETURNS double precision AS $$
DECLARE
     cur refcursor;
     geom1 geometry;
     geom2 geometry;
     temp_geom geometry;
     pi double precision;
     result double precision;
BEGIN
open cur for (select geom_degree from iceberg_clean where
    iceberg = ice_id order by iceberg, date);
```

```
fetch cur into geom1;
fetch cur into temp_geom;
pi := 3.141596;
loop
    geom2 = temp_geom;
    fetch cur into temp_geom;
    exit when not found;
end loop;
result = st_azimuth(geom2, geom1);
result = result * (180/pi);
return result;
close cur;
END;
$$ LANGUAGE 'plpgsql';
```

Listing B.12: Calculate Direction 2

```
CREATE OR REPLACE FUNCTION tr_direction(lat1 double
   precision, lat2 double precision, lon1 double precision,
   lon2 double precision) RETURNS double precision AS $$
DECLARE
    pi double precision;
    result double precision;
    geom_degree1 geometry;
    geom_degree2 geometry;
BEGIN
pi := 3.141596;
geom_degree1 = setsrid(st_makepoint(lon1,lat1),4326);
geom_degree2 = setsrid(st_makepoint(lon2,lat2),4326);
result = st_azimuth(geom_degree1, geom_degree2);
result = result * (180/pi);
return result;
END;
$$ LANGUAGE 'plpgsql';
```

B.3 Trajectory Creation

Listing B.13: Create Trajectory

```
CREATE OR REPLACE FUNCTION tr_create_trajectory_pointm(
   ice_id character varying) RETURNS geometry AS $$
DECLARE
    cur refcursor;
    var1 iceberg_clean%rowtype;
    var2 geometry[];
    geom geometry;
```

```
    i integer;
BEGIN
open cur for (select * from iceberg_clean where iceberg =
   ice_id order by date);
fetch cur into var1;
i := 0;
loop
    var2[i] = var1.pointm;
    fetch cur into var1;
    exit when not found;
    i = i+1;
end loop;
if (array_upper(var2,1)+1) > 1 then
    geom = st_makeline_garray(var2);
end if;
return geom;
close cur;
END;
$$ LANGUAGE 'plpgsql';
```

B.4 Trajectory Summarization

Listing B.14: Count Number of Milestone

```
CREATE OR REPLACE FUNCTION tr_num_milestone(ice_id character
    varying) RETURNS integer AS $$
DECLARE
    cur refcursor;
    var1 geometry;
    var2 geometry[];
    i integer;
BEGIN
open cur for (select geom_degree from table_iceberg3 where
   iceberg = ice_id order by iceberg,date);
fetch cur into var1;
i := 0;
loop
    var2[i] = var1;
    fetch cur into var1;
    exit when not found;
    i = i+1;
end loop;
return (array_upper(var2,1)+1);
close cur;
END;
$$ LANGUAGE 'plpgsql';
```

Listing B.15: Create Table Summary Trajectory

```
CREATE TABLE summary_trajectory (
    iceberg character varying,
    traveled_time integer,
    traveled_dist double precision,
    ave_speed double precision,
    direction double precision,
    trajectory geometry,
    num_milestone integer
);
```

Listing B.16: Create Summary Trajectory

```
CREATE OR REPLACE FUNCTION tr_summary_trajectory() RETURNS
   text AS $$
DECLARE
    data iceberg_temp3%rowtype;
BEGIN
for data in select distinct iceberg from iceberg_clean
loop
    insert into summary_trajectory values (
        data.iceberg,
        tr_traveled_time(data.iceberg),
        tr_traveled_distance(data.iceberg),
        tr_speed(data.iceberg),
        tr_direction(data.iceberg),
        tr_create_trajectory_pointm(data.iceberg),
        tr_num_milestone(data.iceberg)
    );
end loop;
return 'run select * from table_summary_trajectory';
END;
$$ LANGUAGE 'plpgsql';
```

Listing B.17: Create Table Summary Milestone

```
CREATE TABLE table_summary_milestone(
    iceberg character varying,
    first_date integer, first_calender text,
    last_date integer, last_calender text,
    first_position geometry, last_position geometry,
    first_size integer, last_size integer,
    parent_status character varying
)';
```

Listing B.18: Create Summary Milestone

```
CREATE OR REPLACE FUNCTION tr_summary_milestone() RETURNS
    setof refcursor AS $$
DECLARE
    cur refcursor;
    ice_id character varying;
    var1 iceberg_clean%rowtype;
    var2 iceberg_clean%rowtype;
    first_date integer; last_date integer;
    first_position geometry; last_position geometry;
    first_size double precision; last_size double precision;
BEGIN
open cur for (select * from table_iceberg_clean order by
    iceberg, date);
fetch cur into var1;
fetch cur into var2;
loop
    if (var1.iceberg = var2.iceberg) then
        ice_id = var1.iceberg;
        first_date = var1.date;
        last_date = var2.date;
        first_position = var1.geom_metric;
        last_position = var2.geom_metric;
        first_size = var1.total_size_nm;
        last_size = var2.total_size_nm;
        loop
            var1 = var2;
            fetch cur into var2;
            exit when not found;
            if (var1.iceberg = var2.iceberg) then
                last_date = var2.date;
                last_position = var2.geom_metric;
                last_size = var2.total_size_nm;
            else
                exit;
            end if;
        end loop;
        insert into table_summary_milestone values (ice_id,
            first_date, tr_julian_to_calender(first_date),
            last_date, tr_julian_to_calender(last_date),
            first_position, last_position, first_size,
            last_size, null);
    else
        insert into table_summary_milestone values (var1.
            iceberg, var1.date, tr_julian_to_calender(var1.
            date), var1.date, tr_julian_to_calender(var1.date
```

```
            ), var1.geom_metric, var1.geom_metric, var1.
            total_size_nm, var1.total_size_nm, null);
        end if;
        var1 = var2;
        fetch cur into var2;
        exit when not found;
    end loop;
    close cur;
    END;
    $$ LANGUAGE 'plpgsql';
```

Appendix C

Back end Support for Event Detection

C.1 Event Classification

Listing C.1: Create Table Summary Event

```
CREATE TABLE table_summary_event AS SELECT iceberg,
    traveled_dist, num_milestone FROM
    table_summary_trajectory;
ALTER TABLE table_summary_event ADD COLUMN status character
    varying;
```

Listing C.2: Classify Event

```
CREATE OR REPLACE FUNCTION tr_detect_event() RETURNS setof
    refcursor AS $$
DECLARE
     cur refcursor;
     var1 summary_event%rowtype;
     var2 summary_event%rowtype;
     text1 character varying;
     text2 character varying;
     text3 character varying;
BEGIN
open cur for select * from summary_event order by iceberg;
fetch cur into var1;
fetch cur into var2;
loop
text1 = substring(var1.iceberg from 1 for 3);
text2 = substring(var2.iceberg from 1 for 3);
if text1 = text2 then
update table_summary_event set status = 'parent' where
    iceberg = var1.iceberg;
```

```
loop
 if substring(var1.iceberg from 1 for 3) = substring(var2.
    iceberg from 1 for 3) then
        update table_summary_event set status = 'child'
           where iceberg = var2.iceberg;
 else
        exit;
 end if;
 fetch cur into var2;
 exit when not found;
end loop;
else
        if (var1.traveled_dist = 0) or (var1.traveled_dist
           is null) then
                update table_summary_event set status = '
                   grounded' where iceberg = var1.iceberg;
        else
                update table_summary_event set status = '
                   floating' where iceberg = var1.iceberg;
        end if;
end if;
var1 = var2;
fetch cur into var2;
exit when not found;
end loop;
if (var1.traveled_dist = 0) or (var1.traveled_dist is null)
   then
        update table_summary_event set status = 'grounded'
           where iceberg = var1.iceberg;
else
        update table_summary_event set status = 'floating'
           where iceberg = var1.iceberg;
end if;
close cur;
END;
$$ LANGUAGE 'plpgsql';
```

C.2 Calving Detection

Listing C.3: Create Table Summary Calving

```
CREATE TABLE table_summary_calving(
  iceberg character varying,
  first_date integer,
  first_calender text,
  first_size integer,
  first_position geometry,
```

```
        parent character varying,
  calve_date integer,
  calve_calender text,
  parent_size integer,
  calve_position geometry
);
```

Listing C.4: Detect Calving

```
CREATE OR REPLACE FUNCTION tr_detect_calving() RETURNS setof
    refcursor AS $$
DECLARE
  cur refcursor;
  cur2 refcursor;
  var1 temp_family%rowtype;
  var2 temp_family%rowtype;
  first_A_id character varying;
  first_A_date integer;
  first_A_size integer;
  first_A_position geometry;
  shortest_time integer;
  shortest_distance double precision;
  temp_time integer;
  temp_distance double precision;
  parent_id character varying;
  calve_date integer;
  parent_size integer;
  calve_position geometry;
BEGIN
-- Part 1 --
open cur for select * from temp_family where iceberg like '%
   A' order by date;
fetch cur into var1;
first_A_id = var1.iceberg;
first_A_date = var1.date;
first_A_size = var1.total_size_nm;
first_A_position = var1.geom_metric;
close cur;
open cur for select * from temp_family where iceberg =
   substring(first_A_id from 1 for 3) and date <=
   first_A_date order by date desc;
fetch cur into var1;
insert into table_summary_calving values (first_A_id,
   first_A_date, tr_julian_to_calender(first_A_date),
   first_A_size, first_A_position, var1.iceberg, var1.date,
   tr_julian_to_calender(var1.date), var1.total_size_nm,
   var1.geom_metric);
```

```
close cur;
-- Part 2 --
open cur for select * from temp_family where (iceberg, date)
    in (select iceberg, min(date) from temp_family where
   iceberg <> substring(first_A_id from 1 for 3) and iceberg
    <> first_A_id group by iceberg order by iceberg) order
   by iceberg, date asc;
fetch cur into var1;
if (var1.iceberg <> '') then
loop
    open cur2 for select * from temp_family where (iceberg,
       date) in (select iceberg, max(date) from temp_family
               where date < var1.date and total_size_nm >
       var1.total_size_nm group by iceberg order by iceberg)
        order by iceberg, date asc;
        fetch cur2 into var2;
        shortest_time = 999;
loop
temp_time = tr_time_difference(var2.date, var1.date);
temp_distance = st_distance(var1.geom_metric, var2.
   geom_metric);
if  temp_time < shortest_time then
        shortest_time = temp_time;
        parent_id = var2.iceberg;
        calve_date = var2.date;
        parent_size = var2.total_size_nm;
        calve_position = var2.geom_metric;
else
        shortest_distance = st_distance(var1.geom_metric,
           var2.geom_metric);
        if shortest_distance < temp_distance then
                parent_id = var2.iceberg;
                calve_date = var2.date;
                parent_size = var2.total_size_nm;
                calve_position = var2.geom_metric;
        else
                exit;
        end if;
end if;
fetch cur2 into var2;
exit when not found;
end loop;
insert into table_summary_calving values (var1.iceberg, var1
   .date, tr_julian_to_calender(var1.date), var1.
   total_size_nm, var1.geom_metric, parent_id, calve_date,
   tr_julian_to_calender(calve_date), parent_size,
   calve_position);
```

```
close cur2;
fetch cur into var1;
exit when not found;
end loop;
close cur;
else close cur;
end if;
END;
$$ LANGUAGE 'plpgsql';
```

Listing C.5: Create Summary Calving

```
CREATE OR REPLACE FUNCTION tr_summary_calving() RETURNS
   setof refcursor AS $$
DECLARE
        cur refcursor;
        cur2 refcursor;
        temp_check table_summary_event%rowtype;
        check table_summary_event%rowtype;
        data iceberg_clean%rowtype;
        my_val integer;
BEGIN
open cur for select * from table_summary_event order by
   iceberg;
fetch cur into check;
loop
if (check.parent_status = 'parent') then
 open cur2 for select * from table_summary_event where
    iceberg = check.iceberg;
 fetch cur2 into temp_check;
 execute 'create table temp_family(iceberg character varying
    , date integer, latitude double precision, longitude
    double precision, size character varying, geom_degree
    geometry, geom_metric geometry)';
 for data in select * from table_iceberg_clean where iceberg
     like temp_check.iceberg || '%'
loop
        insert into temp_family values (data.iceberg, data.
           date, data.latitude, data.longitude, data.size,
           null, null);
end loop;
execute 'update temp_family set geom_degree = setsrid(
   makepoint(longitude, latitude), 4326)';
execute 'update temp_family set geom_metric = Transform(
   geom_degree, 3031)';
execute 'alter table temp_family add column total_size_nm
   integer';
```

```
execute 'update temp_family set total_size_nm = cast(
   substring(size from 1 for 2) as int) * cast(substring(
   size from 4 for 2) as int)';
execute 'select * from tr_detect_calving()';
execute 'drop table temp_family';
close cur2;
else
        my_val = 1;
end if;
fetch cur into check;
exit when not found;
end loop;
close cur;
END;
$$ LANGUAGE 'plpgsql';
```

Appendix D

Back end Support for Trajectory Data Mining

D.1 Data Interpolation

Listing D.1: Create Table Interpolation

```
CREATE TABLE table_remo_interpolation(
    iceberg character varying,
    date integer,
    calender text,
    latitude double precision,
    longitude double precision,
    flag character varying
);
```

Listing D.2: Interpolation

```
CREATE OR REPLACE FUNCTION tr_remo_interpolation(ice_id
   character varying) RETURNS text AS $$
DECLARE
        cur refcursor;
        var1 table_iceberg_clean%rowtype;
        var2 table_iceberg_clean%rowtype;
        time_differ integer;
        new_time integer; new_x double precision; new_y
           double precision;
        repetition integer; i integer; new_time_string text;
BEGIN
open cur for select * from table_iceberg_clean where iceberg
    = ice_id order by date;
fetch cur into var1;
fetch cur into var2;
loop
```

```
insert into table_remo_interpolation values (var1.iceberg,
   var1.date, tr_julian_to_calender(var1.date), var1.
   latitude, var1.longitude, 'observed');
time_differ = tr_time_difference(var1.date, var2.date);
if time_differ > 1 then
        new_time = var1.date + 1;
        new_x = ((var2.longitude - var1.longitude) / (var2.
           date - var1.date)) + var1.longitude;
        new_y = ((var2.latitude - var1.latitude) / (var2.
           date - var1.date)) + var1.latitude;
        insert into table_remo_interpolation values (var1.
           iceberg, new_time, tr_julian_to_calender(new_time
           ), new_y, new_x, 'derived');
        new_time = new_time + 1;
while (tr_time_difference(new_time, var2.date) > 0)
loop
 if tr_is_leap_year(cast(substring(cast(new_time as text)
    from 1 for 4) as int)) = true then
        if (cast(substring(cast(new_time as text) from 5 for
            3) as int) <= 366) then
                new_x = ((var2.longitude - new_x) / (var2.
                   date - new_time)) + new_x;
                new_y = ((var2.latitude - new_y) / (var2.
                   date - new_time)) + new_y;
                insert into table_remo_interpolation values
                   (var1.iceberg, new_time,
                   tr_julian_to_calender(new_time), new_y,
                   new_x, 'derived');
        else
                new_time_string = cast(cast(substring(cast(
                   new_time as text) from 1 for 4) as int) +
                    1 as text) ||      cast('001' as text);
                new_time = cast(new_time_string as int);
                new_x = ((var2.longitude - new_x) / (var2.
                   date - new_time)) + new_x;
                new_y = ((var2.latitude - new_y) / (var2.
                   date - new_time)) + new_y;
                insert into table_remo_interpolation values
                   (var1.iceberg, new_time,
                   tr_julian_to_calender(new_time), new_y,
                   new_x, 'derived');
  end if;
  new_time = new_time + 1;
 else
        if (cast(substring(cast(new_time as text) from 5 for
            3) as int) <= 365) then
```

```
          new_x = ((var2.longitude - new_x) / (var2.date -
             new_time)) + new_x;
          new_y = ((var2.latitude - new_y) / (var2.date -
             new_time)) + new_y;
          insert into table_remo_interpolation values (var1.
             iceberg, new_time, tr_julian_to_calender(
             new_time), new_y, new_x, 'derived');
        else
          new_time_string = cast(cast(substring(cast(new_time
              as text) from 1 for 4) as int) + 1 as text) ||
             cast('001' as text);
          new_time = cast(new_time_string as int);
          new_x = ((var2.longitude - new_x) / (var2.date -
             new_time)) + new_x;
          new_y = ((var2.latitude - new_y) / (var2.date -
             new_time)) + new_y;
          insert into table_remo_interpolation values (var1.
             iceberg, new_time, tr_julian_to_calender(
             new_time), new_y, new_x, 'derived');
        end if;
        new_time = new_time + 1;
end if;
end loop;
end if;
var1 = var2;
fetch cur into var2;
exit when not found;
end loop;
insert into table_remo_interpolation values (var1.iceberg,
   var1.date, tr_julian_to_calender(var1.date), var1.
   latitude, var1.longitude, 'observed');
close cur;
return 'run select * from table_remo_interpolation';
END;
$$ language 'plpgsql';
```

Listing D.3: Generate Interpolation

```
CREATE OR REPLACE FUNCTION
   tr_remo_generate_interpolation_all() RETURNS text AS $$
DECLARE
        data table_summary_trajectory%rowtype;
BEGIN
for data in select iceberg from table_summary_trajectory
   where num_milestone > 1 and traveled_dist > 0
loop
```

```
        execute 'select * from tr_remo_interpolation( ' ||
           quote_literal(data.iceberg) || ')';
end loop;
execute 'alter table table_remo_interpolation add column
   geom_degree geometry';
execute 'update table_remo_interpolation set geom_degree =
   setsrid(makepoint(longitude, latitude), 4326)';
execute 'alter table table_remo_interpolation add column
   geom_metric geometry';
execute 'update table_remo_interpolation set geom_metric =
   Transform(geom_degree, 3031)';
return 'run select * from table_remo_interpolation';
END;
$$ LANGUAGE 'plpgsql';
```

D.2 Data Classification

Listing D.4: Create Table Classification

```
CREATE TABLE table_remo_classification(
    iceberg character varying,
    date integer,
    calender text,
    speed double precision,
    speed_class integer,
    direction double precision,
    direction_class integer,
    geom_degree geometry)
```

Listing D.5: Data Classification

```
CREATE OR REPLACE FUNCTION tr_remo_classification() RETURNS
   text as $$
declare
        cur refcursor;
        var1 table_remo_interpolation%rowtype;
        var2 table_remo_interpolation%rowtype;
        prev table_remo_interpolation%rowtype;
        speed_value double precision;
        speed_class integer;
        direction_value double precision;
        direction_class integer;
        pi double precision;
BEGIN
open cur for select * from table_remo_interpolation order by
    iceberg, date;
```

```
fetch cur into var1;
fetch cur into var2;
pi = 3.14;
loop
        if (var1.iceberg = var2.iceberg) then
                speed_value = tr_average_speed(var1.date,
                   var1.geom_metric, var2.date, var2.
                   geom_metric);
                direction_value = st_azimuth(var1.
                   geom_metric, var2.geom_metric);
                direction_value = direction_value * (180/pi)
                   ;
        else
                speed_value = tr_average_speed(prev.date,
                   prev.geom_metric, var1.date, var1.
                   geom_metric);
                direction_value = st_azimuth(var1.
                   geom_metric, var2.geom_metric);
                direction_value = direction_value * (180/pi)
                   ;
        end if;
        insert into table_remo_classification values (var1.
           iceberg, var1.date, tr_julian_to_calender(var1.
           date),
        speed_value, tr_remo_assign_class_speed(speed_value)
           , direction_value, tr_remo_assign_class_direction
           (direction_value), var1.geom_degree);
        prev = var1;
        var1 = var2;
        fetch cur into var2;
        exit when not found;
end loop;
insert into table_remo_classification values (var1.iceberg,
   var1.date, tr_julian_to_calender(var1.date), speed_value,
    tr_remo_assign_class_speed(speed_value), direction_value
   , tr_remo_assign_class_direction(direction_value), var1.
   geom_degree);
close cur;
return 'run select * from table_remo_classification';
END;
$$ LANGUAGE 'plpgsql';
```

Listing D.6: Direction Classification

```
CREATE OR REPLACE FUNCTION tr_remo_assign_class_direction(
   direction double precision)
RETURNS integer AS $$
```

```
DECLARE
        classification integer;
BEGIN
if ((direction >= -22.5) and (direction < 22.5)) or (
   direction is null) then
        classification = 0;
elsif (direction >= 22.5) and (direction < 67.5) then
        classification = 45;
elsif (direction >= 67.5) and (direction < 112.5) then
        classification = 90;
elsif (direction >= 112.5) and (direction <= 157.5) then
        classification = 135;
elsif (direction >= 157.5) and (direction < -157.5) then
        classification = 180;
elsif (direction >= -157.5) and (direction < -112.5) then
        classification = 225;
elsif (direction >= -112.5) and (direction < -67.5) then
        classification = 270;
else
        classification = 315;
end if;
return classification;
END;
$$ LANGUAGE 'plpgsql';
```

Listing D.7: Create Summary Milestone

```
CREATE OR REPLACE FUNCTION tr_remo_assign_class_speed(speed
   double precision)
RETURNS integer AS $$
DECLARE
        classification integer;
BEGIN
if (speed = 0) then
        classification = 1;
elsif (speed > 0) and (speed <= 0.2) then
        classification = 2;
elsif (speed > 0.2) and (speed <= 0.4) then
        classification = 3;
elsif (speed > 0.4) and (speed <= 1) then
        classification = 4;
elsif (speed > 1) and (speed <= 2) then
        classification = 5;
elsif (speed > 2) and (speed <= 4) then
        classification = 6;
elsif (speed > 4) and (speed <= 7) then
        classification = 7;
```

```
else
        classification = 8;
end if;
return classification;
END;
$$ LANGUAGE 'plpgsql';
```

D.3 Matrix Generation

Listing D.8: Create Table Matrix

```
CREATE TABLE table_remo_matrix(
    iceberg character varying,
    date integer,
    calender text,
    speed_class integer,
    direction_class integer,
    geom_degree geometry
);
```

Listing D.9: Create Matrix

```
CREATE OR REPLACE FUNCTION tr_remo_matrix(time_start text,
   time_end text)
returns text AS $$
DECLARE
        cur refcursor;
        var1 table_remo_classification%rowtype;
        data table_remo_classification%rowtype;
        time_from integer;
        time_to integer;
        temp_begin integer;
        temp_end integer;
BEGIN
time_from = tr_calender_to_julian(time_start);
time_to = tr_calender_to_julian(time_end);
for data in select distinct iceberg from
   table_remo_classification
loop
        open cur for select * from table_remo_classification
            where iceberg = data.iceberg order by date;
        fetch cur into var1;
        temp_begin = var1.date;
        loop
                temp_end = var1.date;
                fetch cur into var1;
```

```
                exit when not found;
        end loop;
        close cur;
        if (temp_begin <= time_from) and (temp_end >=
           time_to) then
                open cur for select * from
                   table_remo_classification where (iceberg
                   = data.iceberg) and (date >= time_from)
                   and (date <= time_to) order by date;
                fetch cur into var1;
                loop
                        insert into table_remo_matrix values
                            (var1.iceberg, var1.date,
                           tr_julian_to_calender(var1.date),
                            var1.speed_class, var1.
                           direction_class, var1.geom_degree
                           );
                        fetch cur into var1;
                        exit when not found;
                end loop;
                close cur;
        end if;
end loop;
return 'run select * from table_remo_matrix';
END;
$$ LANGUAGE 'plpgsql';
```

D.4 Pattern Detection

Listing D.10: Create Table Concurrence

```
CREATE TABLE table_remo_concurrence(
        iceberg character varying,
    date integer,
    calender text,
    direction_class integer
);
```

Listing D.11: Detect Concurrence

```
CREATE OR REPLACE FUNCTION tr_remo_concurrence(class_search
   integer, time_search text) RETURNS text as $$
DECLARE
        data table_remo_matrix%rowtype;
        date_search integer;
BEGIN
```

```
date_search = tr_calender_to_julian(time_search);
for data in select * from table_remo_matrix where
   direction_class = class_search and date = date_search
   order by iceberg
loop
        insert into table_remo_concurrence values (data.
           iceberg, data.date, tr_julian_to_calender(data.
           date), data.direction_class);
end loop;
return 'run select * from table_remo_concurrence';
end;
$$ language 'plpgsql';
```

Listing D.12: Create Table Constancy

```
CREATE TABLE table_remo_constancy(
        iceberg character varying,
    date integer,
    calender text,
    direction_class integer);
```

Listing D.13: Detect Constancy

```
CREATE OR REPLACE FUNCTION tr_remo_constancy(class_search
   integer, length integer) RETURNS text AS $$
DECLARE
        cur refcursor;
        var1 table_remo_matrix%rowtype;
        i integer;
        temp_date integer[];
        temp_name character varying;
BEGIN
open cur for select * from table_remo_matrix order by
   iceberg, date;
fetch cur into var1;
i = 1;
temp_date = null;
temp_name = var1.iceberg;
loop
        if var1.iceberg = temp_name then
                if var1.direction_class = class_search then
                        temp_date[i] = var1.date;
                        i = i + 1;
                else
                        if array_upper(temp_date,1) >=
                           length then
```

```
                                        for i in 1..array_upper(
                                           temp_date,1)
                                        loop
                                                insert into
                                                   table_remo_constancy
                                                    values (var1.
                                                   iceberg,
                                                   temp_date[i],
                                                   tr_julian_to_calender
                                                   (temp_date[i]),
                                                   class_search);
                                        end loop;
                                        temp_date = null;
                                        i = 1;
                                else
                                        i = 1;
                                        temp_date = null;
                                end if;
                        end if;
                else
                        if array_upper(temp_date,1) >= length then
                                for i in 1..array_upper(temp_date,1)
                                        loop
                                                insert into
                                                   table_remo_constancy
                                                    values (var1.
                                                   iceberg,
                                                   temp_date[i],
                                                   tr_julian_to_calender
                                                   (temp_date[i]),
                                                   class_search);
                                        end loop;
                                        temp_date = null;
                                        i = 1;
                        else
                                i = 1;
                                temp_date = null;
                        end if;
                        temp_name = var1.iceberg;
                        if var1.direction_class = class_search then
                                temp_date[i] = var1.date;
                                i = i + 1;
                        end if;
                end if;
                fetch cur into var1;
                exit when not found;
        end loop;
```

```
close cur;
return 'run select * from table_remo_constancy';
END;
$$ LANGUAGE 'plpgsql';
```

Listing D.14: Create Table Temp Conformity

```
create table table_temp_conformity(
        oid integer,
    iceberg character varying,
    date integer[]);
```

Listing D.15: Create Table Conformity

```
create table table_remo_conformity_direction(
        iceberg character varying,
    date integer[]);
```

Listing D.16: Detect Conformity

```
CREATE OR REPLACE FUNCTION tr_remo_conformity() RETURNS text
    AS $$
DECLARE
        cur1 refcursor;
        cur2 refcursor;
        var1 table_remo_constancy_direction%rowtype;
        var2 table_remo_constancy_direction%rowtype;
        var3 table_temp_conformity_direction%rowtype;
        var4 table_temp_conformity_direction%rowtype;
        i integer;
        temp_name character varying;
        temp_date integer[];
        oid integer;
BEGIN
open cur1 for select * from table_remo_constancy_direction
   order by iceberg, date;
fetch cur1 into var1;
fetch cur1 into var2;
i = 1;
temp_name = var1.iceberg;
temp_date = null;
oid = 1;
loop
        if (var1.iceberg = temp_name) and (var2.iceberg =
           temp_name) and (tr_time_difference(var1.date,
           var2.date) = 1) then
        temp_date[i] = var1.date;
```

```
                i = i + 1;
        else
                temp_date[i] = var1.date;
                if array_upper(temp_date,1) > 1 then
                        insert into
                           table_temp_conformity_direction
                           values (oid, var1.iceberg,
                           temp_date);
                        oid = oid + 1;
                        temp_date = null;
                        i = 1;
                else
                        temp_date = null;
                        i = 1;
                end if;
                temp_name = var2.iceberg;
        end if;
        var1 = var2;
        fetch cur1 into var2;
        exit when not found;
end loop;
temp_date[i] = var1.date;
if array_upper(temp_date,1) > 1 then
        insert into table_temp_conformity_direction values (
           oid, var1.iceberg, temp_date);
end if;
close cur1;
open cur1 for select * from table_temp_conformity_direction
   order by oid;
fetch cur1 into var3;
loop
        open cur2 for select * from
           table_temp_conformity_direction where oid >= var3
           .oid order by oid;
        fetch cur2 into var4;
        loop
                if array_upper(var3.date,1) = array_upper(
                   var4.date,1) then
                        if var3.date[1] = var4.date[1] then
                                insert into
                                   table_remo_conformity_direction
                                    values (var3.iceberg,
                                   var3.date);
                        insert into
                           table_remo_conformity_direction
                           values (var4.iceberg, var4.date);
                        end if;
```

```
                end if;
                fetch cur2 into var4;
                exit when not found;
        end loop;
        close cur2;
        fetch cur1 into var3;
        exit when not found;
end loop;
close cur1;
return 'run select * from table_remo_conformity_direction';
end;
$$ language 'plpgsql';
```

Appendix E

KML File Generation

Listing E.1: KML File Generation using Python

```
def kml():
    f1 = open("quad_a.txt", "r")
    f2 = open("quad_a.kml", "w")
    line=f1.readline()
    line=line.split()
    f2.write("<kml>\n")
    f2.write("<Document>\n")
    for i in range(0,95):   # total number of iceberg
        f2.write("<Placemark>\n")
        f2.write("<name>")
        f2.write(line[0])
        f2.write("</name>\n")
        f2.write("<Style>\n")
        f2.write("<LineStyle>\n")
        f2.write("<color>")
        f2.write("7f0000ff")
        f2.write("</color>\n")
        f2.write("<width>3</width>")
        f2.write("</LineStyle>\n")
        f2.write("</Style>\n")
        #TimeSpan
        oldtime=line[1].split(",")
        f2.write("<TimeSpan>\n")
        f2.write("<begin>")
        f2.write(oldtime[2])
        f2.write("</begin>\n")
        newtime=line[-1].split(",")
        f2.write("<end>")
        f2.write(newtime[2])
        f2.write("</end>\n")
        f2.write("</TimeSpan>\n")
```

```
        #LineString
        f2.write("<LineString>\n<coordinates>\n")
        for j in range (1,len(line)):
            linestr=line[j].split(",")
            f2.write(linestr[0])
            f2.write(",")
            f2.write(linestr[1])
            f2.write("\n")
        f2.write("</coordinates>\n</LineString>\n</Placemark
           >\n")
        line=f1.readline()
        line=line.split()
    f1.close()
    f1 = open("quad_a.txt", "r")
    line=f1.readline()
    line=line.split()
    for e in range(0,95):
        #Point
        for c in range (1,len(line)):
            f2.write("<Placemark>\n")
            f2.write("<name>")
            f2.write(line[0])      #attention!!
            f2.write("</name>\n")
            f2.write("<TimeStamp>")
            f2.write("<when>")
            oldtime=line[c].split(",")
            f2.write(oldtime[2])     #attention!!
            f2.write("</when>\n")
            f2.write("</TimeStamp>\n")
            f2.write("<Point>\n")
            f2.write("<coordinates>")
            f2.write(oldtime[0])
            f2.write(",")
            f2.write(oldtime[1])
            f2.write("\n")
            f2.write("</coordinates></Point>\n")
            f2.write("</Placemark>\n")
        line=f1.readline()
        line=line.split()
    f2.write("</Document>\n")
    f2.write("</kml>")
    f2.close()
```