



UNIVERSIDADE
FEDERAL
DE PERNAMBUCO



UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

INCORPORANDO DADOS DE TRAJETÓRIA NO PostgreSQL: IMPLEMENTAÇÃO DO TIPO ABSTRATO DE DADOS *TRAJECTORY*

Trabalho de Graduação em Ciência da Computação
2013.2

Aluno: Paulo Eduardo do Nascimento Carvalho (penc@cin.ufpe.br)

Orientadora: Valéria Cesário Times (vct@cin.ufpe.br)

Recife, março de 2014

Paulo Eduardo do Nascimento Carvalho

**INCORPORANDO DADOS DE TRAJETÓRIA NO PostgreSQL:
IMPLEMENTAÇÃO DO TIPO ABSTRATO DE DADOS *TRAJECTORY***

Monografia apresentada como requisito parcial para a obtenção do Grau de Bacharel em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco.

Orientadora: VALÉRIA CESÁRIO TIMES

Recife, março de 2014

Assinaturas

Monografia apresentada como requisito parcial para a obtenção do Grau de Bacharel em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco.

Paulo Eduardo do Nascimento Carvalho

Aluno

Valéria Cesário Times

Orientadora

Recife, março de 2014

*“Faça as coisas o mais simples que você puder,
porém não se restrinja às mais simples.”*

Albert Einstein

Agradecimentos

Primeiramente, agradeço a Deus por ter me proporcionado saúde e força.

Agradeço à toda minha família, em especial aos meus pais, Ladjane e Genival Carvalho, por todos os princípios e valores ensinados e por todo apoio e dedicação à minha educação. Sem eles eu não teria uma base sólida para conseguir enfrentar todos os desafios da vida.

Aos professores do Centro de Informática da Universidade Federal de Pernambuco pelas cobranças e pelo conhecimento transmitido, em especial à professora Valéria Cesário Times, que mesmo estando longe, esteve bastante presente na produção deste trabalho, com muita atenção e paciência, e também aos professores Fernando Fonseca, Sílvio Melo, Ruy de Queiroz e Márcio Cornélio, pela oportunidade de ampliar e compartilhar meu conhecimento através das atividades de monitoria.

Não posso deixar de agradecer aos meus amigos do Centro de Informática. Eles, mais do que ninguém, sabem todo o esforço que foi necessário desde o primeiro período e, compartilharam comigo, ao longo do curso, aperreios, noites em claro, descontrações e muita dedicação.

Muito obrigado!

Índice de Figuras

Figura 2.1 – Exemplo de uma trajetória com 24 pontos espaço-temporais.....	6
Figura 2.2 – Stops encontrados utilizando o método IB-SMoT [1].....	7
Figura 2.3 – Exemplo de candidato a stop encontrado através do método CB-SMoT.	8
Figura 4.1 – Modelo Entidade-Relacionamento para a aplicação	20

Índice de Tabelas

Tabela 3.1 – Funções necessárias para definição dos tipos Position e Trajectory ...	11
Tabela 3.2 – Representação textual para os tipos propostos.	12
Tabela 4.1 – Amostra dos dados utilizados	20

Resumo

Com o crescimento e a popularização de GPS e outros dispositivos capazes de capturar informações de posicionamento global de objetos móveis, existe uma demanda de armazenamento e análise de trajetórias de objetos móveis. Porém, em um SGBD geográfico de dados vetoriais, tipicamente são armazenadas informações geográficas do tipo: geometria, ponto, linha e polígono. Este trabalho traz a proposta de especificação e implementação de um novo tipo abstrato de dados (TAD), chamado *Trajectory*, em um SGBD de código aberto, chamado PostgreSQL, de maneira que o tipo de dado proposto possa ser utilizado como qualquer outro tipo existente no SGBD. Este trabalho também inclui a implementação de uma aplicação que utiliza dados de trajetórias, a fim de mostrar para fins de validação, o armazenamento, a manipulação e a consulta sobre o novo tipo de dado.

Palavras-chave: SGBD, Banco de dados, banco de dados geográfico, trajetória, tipo abstrato de dados.

Abstract

With the growth and popularization of GPS and other devices capable of capturing global positioning information of moving objects, there is a need for the storage and analysis of trajectories of moving objects. However, in a geographical DBMS designed for the management of vector data, the typically stored geographic information are: geometry, point, line and polygon. This work presents a proposal to specify and implement a new abstract data type (ADT), called Trajectory, in an open source DBMS, called PostgreSQL, so that the proposed data type can be used just like any other type existing in the DBMS. The proposed will be follow by the implementation of an application where use the trajectory data. This work includes the implementation of an application that uses trajectory data in order to validate and demonstrate the storage, manipulation and query over this new data type.

Keywords: DBMS, Databases, geographic databases, trajectory, abstract data type.

Sumário

1) Capítulo 1: Introdução	1
1.1) Contextualização	1
1.2) Motivação	2
1.3) Objetivos.....	2
1.4) Estrutura do Documento	3
2) Capítulo 2: Fundamentação Teórica	4
2.1) Introdução.....	4
2.2) Conceitos Básicos	4
2.2.1) Tipo Abstrato de Dado (TAD).....	4
2.2.2) Criação de TADs no PostgreSQL	4
2.2.3) Trajetória.....	6
2.2.4) Adição de Semântica à Trajetória	7
2.3) Trabalhos Relacionados	8
2.4) Considerações finais	9
3) Capítulo 3: Tipo Abstrato de Dado <i>Trajectory</i>	10
3.1) Introdução.....	10
3.2) Implementação	10
3.2.1) Definição do tipo <i>Trajectory</i> no PostgreSQL	12
3.2.2) Extensão do PostGIS.....	17
3.2.3) Adicionando semântica ao tipo <i>Trajectory</i>	18
4) Capítulo 4: Validação do TAD proposto	20
4.1) Introdução.....	20
4.2) Aplicação utilizando o TAD proposto	20
4.3) Conclusão.....	23
5) Capítulo 5: Conclusão	24

5.1) Introdução.....	24
5.2) Principais Contribuições	24
5.3) Trabalhos Futuros	24
Referências Bibliográficas	25
Apêndice A - Scripts na linguagem C.....	27
Apêndice B - Scripts na linguagem PL/pgSQL	39

1) Capítulo 1: Introdução

1.1) Contextualização

Entre os mais diversos ramos de atividade, a grande maioria das empresas demanda hoje de algum tipo de armazenamento e gerenciamento de dados. Bancos de dados são soluções comumente utilizadas para esse fim, devido à forma estruturada de armazenamento, sempre com o objetivo de facilitar a consulta, inserção e remoção de dados.

Entre os Sistemas de Gerenciamento de Banco de Dados (SGBD) há um tipo que é capaz de lidar com dados geográficos, os quais podem ser armazenados no formato *raster* (denotado por uma matriz de células de tamanho regular) ou no formato vetorial (representado por um conjunto de objetos identificáveis que podem ocupar a mesma posição espacial). No entanto, o foco desse trabalho é sobre Sistema de Gerenciamento de Banco de Dados Geográfico (SGBDG) de dados vetoriais, que é capaz de armazenar informações espaciais através do armazenamento e da manipulação de geometrias, representadas por pontos, linhas e polígonos.

Com o crescimento e popularização de GPS (Global Positioning Systems) e outros dispositivos capazes de capturar informações de posicionamento global de objetos móveis, surge a necessidade de armazenamento e análise de trajetórias de objetos móveis. Nesse cenário, a localização espacial do objeto está associada também a uma informação temporal. Uma vez que essa informação espaço-temporal é coletada, é possível obter a trajetória realizada pelo objeto, onde é possível ainda extrair e/ou aproximar diversas outras informações como: distância total percorrida, velocidade mínima, média e máxima com que o objeto se deslocou. Nesse sentido, a pesquisa sobre modelagem de trajetória é cada vez mais importante, a fim facilitar a consulta, inserção, remoção e análise de dados desse tipo.

1.2) Motivação

Existem inúmeras aplicações de sistemas de informações geográficas que demandam a representação de dados espaço-temporais, e em particular, a modelagem e manipulação de dados de trajetórias. Porém, os SGBDs espaciais disponíveis no mercado não permitem o uso do dado de trajetória como qualquer outro tipo de dado disponível pelo sistema de banco de dados, i.e. como um *Tipo Abstrato de Dado*.

Com o aumento e a popularização de dispositivos capazes de capturar informações geográficas de posicionamento em tempo real, tornou-se fácil obter dados de objetos em movimento, e tal representação espaço-temporal pode ser capaz de prover o suporte necessário para o armazenamento e a manipulação de dados espaço-temporais de diversas aplicações, tais como rastreamento de entregas de mercadorias, gerenciamento do derramamento de óleo no oceano, e monitoramento de frotas de veículos.

Os dispositivos capazes de capturar informações de posicionamento coletam apenas informações sobre a trajetória percorrida pelo objeto móvel e tais informações são chamadas na literatura de trajetória bruta [7]. Porém, existe a necessidade de extrair conhecimento ou informações semânticas com base nesta trajetória bruta. Por exemplo, pode ser útil para um dado domínio de aplicação, saber se o objeto estava parado ou em movimento, em que rua, bairro ou cidade o objeto estava, e assim por diante. Dessa forma, existe a necessidade de armazenar a trajetória bruta que pode ser processada para extrair esse tipo de informação.

1.3) Objetivos

Em um SGBD geográfico, tipicamente são armazenadas informações geográficas do tipo: geometria, ponto, linha e polígono. Este trabalho objetiva a especificação e implementação de um novo tipo abstrato de dados, chamado *Trajectory*, em um SGBD de código aberto, chamado PostgreSQL, de maneira que o tipo de dado proposto possa ser utilizado como qualquer outro tipo existente no SGBD.

Como SGBD foi escolhido o PostgreSQL, que é um SGBD extensível com código fonte aberto que permite a adição de novos tipos de dados definidos pelo usuário [11]. Além de ser código aberto e ter uma comunidade bastante ativa, existe uma extensão chamada PostGIS, que dá suporte ao PostgreSQL à dados geográficos, o qual também é de código aberto.

O desenvolvimento do tipo abstrato de dados *Trajectory* engloba a proposta de uma forma de armazenamento interna para os dados de trajetória além de modificações na camada de processamento de consultas espaciais do SGBD PostgreSQL/PostGIS para permitir o tratamento de relacionamentos topológicos envolvendo dados de trajetória.

É pretendido também validar o tipo de dado proposto, por meio da implementação de uma aplicação utilizando dados de trajetória reais para exemplificação do armazenamento, da manipulação e da consulta com base no tipo *Trajectory* proposto.

1.4) Estrutura do Documento

Esse documento está dividido em cinco capítulos e os demais capítulos estão organizados como segue. O capítulo 2 apresentará os conceitos básicos, seguido pelo Capítulo 3 que trará as especificações e os detalhes do tipo de dado implementado. Por fim, o Capítulo 4 apresentará uma aplicação prática utilizando o tipo de dado proposto seguido pelo Capítulo 5 que listará as principais contribuições e os possíveis trabalhos futuros.

2) Capítulo 2: Fundamentação Teórica

2.1) Introdução

Este capítulo tem como finalidade proporcionar ao leitor um apanhado dos principais conceitos das áreas de pesquisa relacionadas ao estudo realizado para então facilitar o entendimento do problema que se pretende resolver neste trabalho. Para isso, a Seção 2.2 lista os conceitos básicos sobre TADs, PostgreSQL, trajetória e enriquecimento semântico da trajetória. Já na Seção 2.3 são expostos alguns trabalhos relacionados. Por fim, a Seção 2.4 encerra este capítulo.

2.2) Conceitos Básicos

Esta seção lista os conceitos básicos necessários ao entendimento do trabalho realizado. Ela contém a definição de um tipo abstrato de dados (Seção 2.2.1), e explica como é possível criar um TAD no PostgreSQL (Seção 2.2.2). Além disso, uma definição formal de trajetória é listada na Seção 2.2.3 e na Seção 2.2.4, é descrito brevemente como é possível adicionar semântica às trajetórias brutas extraídas de dispositivos de posicionamento.

2.2.1) Tipo Abstrato de Dado (TAD)

Um tipo abstrato de dados pode ser definido como um conjunto de valores de dados e operações associadas, precisamente especificados e independentes de qualquer implementação em particular [3]. Em outras palavras, a forma interna do tipo fica abstraída através de funções de acesso. Um exemplo clássico de TAD é o tipo de dado pilha, para o qual devem ser fornecidas funções para criar uma pilha vazia, para colocar elementos na pilha e para tirar elementos da pilha [2].

2.2.2) Criação de TADs no PostgreSQL

Nesta seção será detalhado como um novo tipo de dado pode ser definido estendendo o SGBD PostgreSQL. Os conceitos descritos serão utilizados como ponto de partida para a definição do tipo proposto neste trabalho, o TAD *Trajectory*.

PostgreSQL é um SGBD extensível com código fonte aberto que permite a adição de novos tipos de dados definidos pelo usuário [11]. Um tipo definido pelo usuário pode ser especificado utilizando a linguagem de baixo nível C ou por meio de procedimentos armazenados e criados por meio da linguagem PL/pgSQL.

O SGBD PostgreSQL dispõe de várias propriedades e funções que definem características sobre como o novo tipo de dados será armazenado e manipulado. As funções de *input* e *output* são obrigatórias para a definição de um novo tipo. A função de *input* é responsável pela conversão da representação textual para representação interna do tipo de dado enquanto que a função *output* realiza a conversão inversa. De forma facultativa, é possível definir funções binárias de *input* e *output*, chamadas de *send* e *receive*. Neste caso, a função *send* é responsável para conversão da representação binária para a representação interna do tipo, ao passo que a função *receive* corresponde à conversão inversa.

Uma propriedade bastante importante, mesmo que não obrigatória, é a propriedade *internallength*, que diz respeito ao tamanho que o novo tipo de dado irá ocupar em memória, i.e. quantos bytes serão necessários para o seu armazenamento. Essa propriedade pode ser variável ou fixa. Quando um tipo possui tamanho variável, é necessário definir um elemento em sua estrutura do tipo inteiro de 4 bytes (*int32*), responsável por armazenar o tamanho do dado.

Em seguida, deve ser definida a estratégia de armazenamento a ser usada pelo SGBD para manter o tipo de dado em memória secundária. O PostgreSQL oferece as seguintes estratégias: *plain*, *extended*, *external* e *main*. A estratégia do tipo *plain* é voltada para tipos de dados que possuem tamanho fixo, onde o armazenamento é feito em linha e não de modo comprimido. Para tipos de tamanho variável, é recomendado o uso da estratégia *extended*, *external* ou *main*. Na estratégia *extended*, primeiro, o sistema tenta comprimir, caso o tamanho comprimido ainda seja muito grande, ele será movido para fora da tabela principal, que é a tabela onde o tipo está definido como coluna. Já no tipo *external*, o sistema permite que o tipo seja movido para fora da tabela sem ter que passar por uma compressão. A estratégia *main* pode ser visto como meio termo entre as estratégias *extended* e *external*. O sistema sempre tentará colocar o valor na tabela principal, permitindo a compressão. Tipos que utilizam essa estratégia têm prioridade em tipos que utilizam as demais estratégias, no momento do armazenamento. Caso ainda assim não seja possível manter o valor na tabela principal, o valor será movido para fora dela.

Concluída a especificação do tipo, é possível definir funções (*CREATE FUNCTION*) para manipulação deste, que podem ser de acesso, i.e. realiza a chamada de uma função definida em baixo nível (linguagem C), ou uma função definida utilizando linguagem de alto nível, i.e PL/pgSQL.

2.2.3) Trajetória

Dado um objeto em movimento, equipado com um sistema receptor de GPS ou outro dispositivo capaz de capturar informações geográficas de posicionamento, a trajetória desse objeto pode ser definida como uma sequência, ordenada pelo tempo, de posições capturadas por esse dispositivo por um certo período de tempo e associadas ao instante de captura. Cada dado coletado deve conter a posição do objeto e o instante de captura do dado.

Dessa forma, uma trajetória pode ser descrita como uma lista de pontos espaço-temporais, ou seja, uma lista de tuplas do tipo (x, y, t) onde (x, y) representa a posição do objeto móvel em um determinado instante t .

Definição 1: Uma trajetória é formada por uma lista de pontos espaço-temporal $(p_0, p_1, \dots, p_N)$ onde $p_i = (x_i, y_i, t_i)$, x_i, y_i e $t_i \in \mathbf{R}$ para $i = 0, 1, \dots, N$ e $t_0 < t_1 < \dots < t_N$



Figura 2.1 – Exemplo de uma trajetória com 24 pontos espaço-temporais

Uma trajetória pode conter um número muito grande de pontos, dependendo do intervalo de rastreamento do objeto. Por exemplo, considere um objeto equipado com um aparelho de GPS que está programado para coletar a posição espacial em intervalos fixos de 5 segundos e por um período de 2 horas. Ao final do intervalo

proposto, serão capturados 1440 pontos. Porém, esses dados coletados consistem em uma *raw trajectory*, ou trajetória bruta.

2.2.4) Adição de Semântica à Trajetória

A grande parte dos trabalhos que envolvem trajetória, consistem em realizar análises a partir da trajetória bruta, a fim de descobrir alguma informação semântica relevante. Existem diversas abordagens que podem ser utilizadas para o enriquecimento da trajetória, porém este trabalho se concentra em um determinado tipo de abordagem, proposta por Spaccapietra [13], onde a trajetória é composta por unidades de *stops* (*paradas do objeto móvel*) e *moves* (*movimentos do objeto móvel*).

A partir da definição conceitual, definida por Spaccapietra [Ref], diversos pesquisadores elaboraram métodos capazes de extrair elementos que representam as unidades de *stops* e *moves*. Alguns desses métodos são expostos a seguir.

O primeiro método, chamado de *Intersection-Based Stops and Moves of Trajectory* (IB-SMoT) [1], baseia-se na interseção da trajetória bruta com feições geográficas de interesse do domínio da aplicação. Os segmentos da trajetória onde há interseção, são denominados de *stops candidatos* da trajetória. Para que um candidato seja de fato tido como um *stop*, é necessário verificar se o tempo de permanência nesse local atinge um certo limiar, definido pelo usuário.

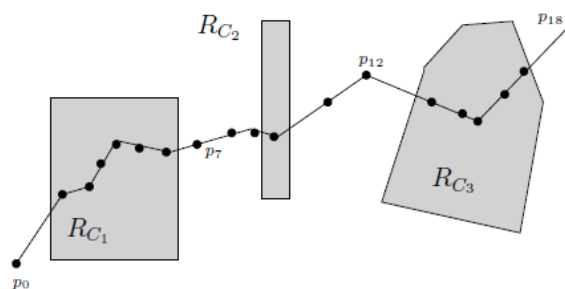


Figura 2.2 – Stops encontrados utilizando o método IB-SMoT [1].

O método chamado de *Clustering-Based Stops and Moves of Trajectory* (CB-SMoT) [8] propõe a localização de *stops* a partir da variação de velocidade do objeto. Esse método identifica clusters de pontos onde o objeto possui velocidade

abaixo de um determinado limiar, durante um intervalo de tempo mínimo. A Figura 2.2 ilustra um *stop* candidato encontrado em uma trajetória.



Figura 2.3 – Exemplo de candidato a *stop* encontrado através do método CB-SMoT.

Um terceiro método, chamado de *Direction-Based Stops and Moves of Trajectory* (DB-SMoT) [12] que cria clusters de forma parecida com o CB-SMoT porém, levando em consideração a variação de direção ao invés da variação de velocidade. Tanto no CB-SMoT quanto no DB-SMoT os clusters obtidos são tidos como *stops candidatos* e para que seja possível identificar quais dos candidatos são realmente um *stop* é necessário realizar sobreposições sobre locais de interesse do domínio, de forma semelhante ao método IB-SMoT.

2.3) Trabalhos Relacionados

Nesta Seção serão apresentados pesquisas com propósito semelhante ao estudo realizado, no que diz respeito ao armazenamento e manipulação de dados de trajetória.

Em HERMES: Aggregative LBS via a Trajectory DB Engine [9] é apresentado um protótipo de que permite o suporte de serviços baseados em localização (*Location-Based Services - LBS*). Para dar suporte inteligente e eficaz a LBS, o sistema mantém um banco de dados de trajetória, que possui funcionalidades interessantes como: consultas baseadas em coordenadas, como consultas envolvendo intervalo, proximidade; consultas baseadas em trajetórias, como topologia, navegação, similaridade; ou consultas baseadas em combinações. HERMES é o primeiro trabalho que apresenta um conjunto completo do que há de

mais avançado em algoritmos de processamento de consultas em banco de dados de trajetória [9].

O principal objetivo do HERMES é dar suporte a modelagem e consulta de objetos que se movem continuamente. Aproveitando interfaces de extensibilidades providas pelos modernos SGBDs Objeto-relacional, HERMES DB é desenvolvido como uma extensão que promove a funcionalidade de trajetória ao SGBD Oracle, definindo um tipo de dados de trajetória e uma coleção de operações.

Já SECONDO [4] apresenta um SGBD extensível ideal para pesquisa, ensino e implementação de sistemas de banco de dados, ideal para dados não convencionais. Nele não há um modelo de dados fixo, sendo aberto para implementação de novos modelos. SECONDO consiste em três elementos: (I) o kernel, que oferece processamento de consultas sobre um conjunto de álgebras implementados, cada um oferecendo a implementação de tipos e operadores, (II) o otimizador, que implementa a parte essencial para uma linguagem baseada em SQL, e (III) a interface gráfica do usuário, que é extensível para visões de novos tipos de dados e que fornece uma maneira de visualização sofisticada para objetos espaciais e espaço-temporais (em movimento).

Exemplos de conjuntos de álgebra implementados no SECONDO são: relações, tipos de dados espaciais, árvores-R e objetos midi (arquivos de música), cada um com suas operações apropriadas. SECONDO pode suportar dados de objetos móveis por meio do uso de objetos espaço-temporais, e provê uma GUI especializada para visualização desses dados [4].

2.4) Considerações finais

Nesse capítulo, foi mostrado a definição de trajetória bruta e como é possível adicionar semântica a ela. No próximo capítulo, será mostrado uma proposta de implementação do tipo de dados *Trajectory* no PostgreSQL e as especificidades implementadas para esse tipo.

3) Capítulo 3: Tipo Abstrato de Dado *Trajectory*

3.1) Introdução

Neste capítulo, será apresentada a proposta do tipo de dado abstrato *Trajectory* e as modificações necessárias que foram realizadas para permitir o tratamento de relacionamentos topológicos envolvendo dados de trajetória.

3.2) Implementação

Conforme indica a **Definição 1**, uma trajetória pode ser representada como sendo um conjunto de pontos espaço-temporais. Desta forma, antes de implementar de fato o TAD *Trajectory*, foi necessário criar um TAD que represente um ponto espaço-temporal, denominado neste trabalho de *Position*. Esse novo tipo de dado foi definido como uma estrutura do tipo *struct* na linguagem C.

A estrutura do tipo de dado *Position* é apresentada no trecho de código exibido na Listagem 3.1, e detalhada como segue:

- O elemento *x* armazena a coordenada *x* do objeto;
- O elemento *y* armazena a coordenada *y* do objeto;
- O elemento *t* armazena o instante que foi coletado a posição do objeto;

```
01: typedef struct Position {  
02:     double x;  
03:     double y;  
04:     Timestamp t;  
05: } Position;
```

Listagem 3.1 – Tipo de Dado *Position*.

De acordo com a estrutura proposta na Listagem 3.1, esse novo tipo de dado possui um tamanho fixo, obtido a partir da soma do tamanho dos elementos que o compõe. De acordo com a documentação do PostgreSQL [11] o tipo de dado *double* utiliza 8 *bytes* para armazenamento e o tipo *Timestamp* também utiliza 8 *bytes*, logo o tamanho necessário para o armazenamento do tipo *Position* é de 24 *bytes*.

Uma vez que foi definido o TAD *Position*, é possível definir o TAD *Trajectory*. A estrutura do tipo de dado *Trajectory* é apresentada no trecho de código exibido na Listagem 3.2, e detalhada como segue:

- O elemento `vl_len` armazena o tamanho do dado, pois este possui tamanho variável;
- O elemento `nposs` armazena a quantidade de posições na trajetória;
- O elemento `p` armazena a estrutura de dados que contém o conjunto de posições da trajetória;

```

01: typedef struct Trajectory {
02:     int32 vl_len_;
03:     int32 nposs;
04:     Position p[1];
05: } Trajectory;

```

Listagem 3.2 – Tipo de Dado *Trajectory*.

Diferentemente do tipo *Position*, o tipo proposto *Trajectory* possui um tamanho variável. Conforme descrito na Seção 2.2.2, um tipo de dado com tamanho variável deve possuir como primeiro elemento um *int32*, responsável por armazenar o tamanho em *bytes* da instância do tipo. Logo, considerando que o tipo *int32* precisa de 4 *bytes* para armazenamento e o tipo *Position* necessita de 24 *bytes*, o tipo *Trajectory* precisará de $8 + 16N$ *bytes*, onde *N* representa o número de *Positions*.

Para que seja possível a criação do tipo proposto no PostgreSQL foi necessário definir as seguintes funções de conversão:

Entrada	Saída	Nome
<i>String</i>	<i>Position</i>	<code>position_in</code>
<i>Position</i>	<i>String</i>	<code>position_out</code>
<i>String</i>	<i>Trajectory</i>	<code>trajectory_in</code>
<i>Trajectory</i>	<i>String</i>	<code>trajectory_out</code>

Tabela 3.1 – Funções necessárias para definição dos tipos *Position* e *Trajectory*

Conforme apresentado na Seção 2.2.2, as funções de *input* (`position_in` e `trajectory_in`) são definidas para conversão da representação textual para a representação interna do tipo de dado, enquanto que as funções de *output* (`position_out` e `trajectory_out`) realizam a conversão inversa.

Nome	Descrição
<i>Position</i>	(x, y, t)
<i>Trajectory</i>	[(x1,y1,t1),...]

Tabela 3.2 – Representação textual para os tipos propostos.

A Tabela 3.2 mostra a representação textual para cada tipo de dado proposto, e exemplos da representação textual são dados a seguir:

- *Position*

'(116.37177,39.91094,2008-02-02 13:39:50)'

- *Trajectory*

'[(116.37177,39.91094,2008-02-02 13:39:50), (116.37177,39.91094,2008-02-02 13:39:51), (116.37177,39.91094,2008-02-02 13:39:52), (116.37177,39.91094,2008-02-02 13:39:54)]'

3.2.1) Definição do tipo *Trajectory* no PostgreSQL

As definições na linguagem C, descritas na Seção 3.2 tornam possível a criação dos tipos de dado *Position* e *Trajectory* utilizando o comando SQL *CREATE TYPE*. Com isso, é necessário definir as características que os tipos de dados possuem, conforme descrito na Seção 2.2.2. Dentro desse contexto, esta seção detalha a criação dos TADs *Position* e *Trajectory* no SGBD PostgreSQL.

Conforme apresentando na Seção 3.2, o tipo *Position* foi definido com tamanho fixo de 24 bytes, logo, de acordo com a Seção 2.2.2, a estratégia utilizada para armazenamento deve ser a estratégia *plain*. Dessa forma, a Listagem 3.3 mostra o código de criação das funções de *input* e *output* e do tipo *Position*:

```

01: CREATE TYPE position;
02:
03: CREATE FUNCTION position_in(cstring) RETURNS position AS
04:     '$libdir\trajectory.so', 'position_in'
05:     LANGUAGE C IMMUTABLE STRICT;
06:
07: CREATE FUNCTION position_out(position) RETURNS cstring AS
08:     '$libdir\trajectory.so', 'position_out'
09:     LANGUAGE C IMMUTABLE STRICT;
10:
11: CREATE TYPE position (
12:     internallength = 24,
```

```

013:     input = position_in,
014:     output = position_out,
015:     storage = plain
016: );

```

Listagem 3.3 – Criação do tipo *Position* no PostgreSQL.

De maneira diferente, o tipo *Trajectory* possui tamanho variável, logo é possível escolher entre as três estratégias detalhadas na Seção 2.2.2: *extended*, *external* ou *main*. A estratégia escolhida foi a *main*, devido à sua prioridade de armazenamento na tabela principal. A Listagem 3.4 mostra o código de criação das funções de *input* e *output* e do tipo *Trajectory*.

```

01: CREATE TYPE trajectory;
02:
03: CREATE FUNCTION trajectory_in(cstring) RETURNS trajectory AS
04:     '$libdir\trajectory.so', 'trajectory_in'
05:     LANGUAGE C IMMUTABLE STRICT;
06:
07: CREATE FUNCTION trajectory_out(trajectory) RETURNS cstring AS
08:     '$libdir\trajectory.so', 'trajectory_out'
09:     LANGUAGE C IMMUTABLE STRICT;
10:
11: CREATE TYPE trajectory (
12:     internallength = VARIABLE,
13:     input = trajectory_in,
14:     output = trajectory_out,
15:     storage = main
16: );

```

Listagem 3.4 – Criação do tipo *Trajectory* no PostgreSQL.

Uma vez que definidos os novos tipos, é possível utilizá-los como colunas na criação de tabelas, conforme é exibido no exemplo da Listagem 3.5.

```

017: CREATE TABLE table_position (
018:     id int,
019:     p position,
020:     PRIMARY KEY(id)
021: );
022:
023: CREATE TABLE table_trajectory (
024:     id int,
025:     t trajectory,
026:     PRIMARY KEY(id)
027: );

```

Listagem 3.5 – Exemplo de uso dos novos tipos.

Com as tabelas criadas é possível inserir, remover, atualizar e/ou consultar dados, conforme é mostrado nos exemplos de aplicação do comando de inserção da Listagem 3.6:

```
01: INSERT INTO table_position
02: VALUES ('(232.10, 2321.22, 2014-12-12 12:00)');
03:
04: INSERT INTO table_trajectory
05: VALUES ('[(232.10, 2321.22, 2014-12-12 12:00:00), (232.10,
    2321.22, 2014-12-12 12:00:30)]');
```

Listagem 3.6 – Exemplo de inserção com os novos tipos.

Além das funções de *input* e *output* para cada novo tipo criado, outras funções foram definidas, utilizando linguagem de baixo nível C, e são detalhadas como segue:

- CREATE FUNCTION to_point(position) RETURNS point
 - Retorna um elemento do tipo *point* construído a partir do tipo *position*, i.e. retorna a parte espacial do tipo *position*.
- CREATE FUNCTION get_timestamp(position) RETURNS Timestamp
 - Retorna um elemento do tipo *Timestamp* construído a partir do tipo *position*, i.e. retorna a parte temporal do tipo *position*.
- CREATE FUNCTION make_position(point, timestamp) RETURNS position
 - Retorna um elemento do tipo *position* construído a partir de um *point* e um *Timestamp*.
- CREATE FUNCTION position_add(position, position) RETURNS position
 - Retorna a soma de dois elementos do tipo *position*. Seja p1 (x1, y1, t1) e p2 (x2, y2, t2), $p1+p2 = (x1+x2, y1+y2, \max(t1, t2))$.
- CREATE FUNCTION position_sub(position, position) RETURNS position
 - Retorna a subtração de dois elementos do tipo *position*. Seja p1 (x1, y1, t1) e p2 (x2, y2, t2), $p1-p2 = (x1-x2, y1-y2, \min(t1, t2))$.
- CREATE FUNCTION npositions(trajectory) RETURNS int
 - Retorna a quantidade de *position* no elemento *trajectory* passado como parâmetro.
- CREATE FUNCTION trajectory_start(trajectory) RETURNS position

- Retorna o primeiro *position* que compõe o elemento *trajectory* passado como parâmetro.
- CREATE FUNCTION trajectory_end(trajectory) RETURNS position
 - Retorna o último *position* que compõe o elemento *trajectory* passado como parâmetro.
- CREATE FUNCTION position_n(trajectory, int) RETURNS position
 - Retorna o n-ésimo *position* que compõe o elemento *trajectory* passado como parâmetro.
- CREATE FUNCTION trajectory_add(trajectory, position) RETURNS trajectory
 - Retorna um *trajectory* composto por todos os *position* que compõe o *trajectory* passado como parâmetro e o *position* passado como parâmetro.
- CREATE FUNCTION segment(trajectory, int, int) RETURNS trajectory
 - Retorna um segmento do *trajectory* passado como parâmetro limitado pelos inteiros passados como parâmetros, i.e. um *trajectory* que começa no n-ésimo *position* e termina no m-ésimo *position* do *trajectory* passado como parâmetro, sendo n e m os inteiros passados como parâmetro.
- CREATE FUNCTION segment(trajectory, timestamp, timestamp) RETURNS trajectory
 - Retorna um segmento do *trajectory* passado como parâmetro limitado pelo intervalo de tempo passado como parâmetros, i.e. um *trajectory* que começa e termina dentro do intervalo passado como parâmetro.

Também foram definidas funções utilizando a linguagem PL/pgSQL, que são detalhas como segue:

- CREATE FUNCTION length(trajectory trajectory) RETURNS float
 - Retorna o tamanho do *trajectory* passado como parâmetro, i.e. a distância total entre o *start* e o *end*. Utiliza a função ST_Length implementada no PostGIS, realizando um *cast* do tipo *trajectory* para o tipo *geography*.
- CREATE FUNCTION avg_length(trajectory trajectory) RETURNS float

- Retorna a média de distância entre dois *positions* consecutivos do *trajectory* passado como parâmetro.
- CREATE FUNCTION top_length(trajectory trajectory) RETURNS float
 - Retorna a maior distância entre dois *positions* consecutivos do *trajectory* passado como parâmetro.
- CREATE FUNCTION low_length(trajectory trajectory) RETURNS float
 - Retorna a menor distância entre dois *positions* consecutivos do *trajectory* passado como parâmetro.
- CREATE FUNCTION duration(trajectory trajectory) RETURNS float
 - Retorna a duração total em segundos do *trajectory* passado como parâmetro.
- CREATE FUNCTION top_duration(trajectory trajectory) RETURNS float
 - Retorna a maior duração em segundos entre dois *positions* consecutivos do *trajectory* passado como parâmetro.
- CREATE FUNCTION low_duration(trajectory trajectory) RETURNS float
 - Retorna a menor duração em segundos entre dois *positions* consecutivos do *trajectory* passado como parâmetro.
- CREATE FUNCTION avg_duration(trajectory trajectory) RETURNS float
 - Retorna a média de duração em segundos entre dois *positions* consecutivos do *trajectory* passado como parâmetro.
- CREATE FUNCTION avg_speed(trajectory trajectory) RETURNS float
 - Retorna a velocidade média do *trajectory* passado como parâmetro, em quilômetros por hora.
- CREATE FUNCTION top_speed(trajectory trajectory) RETURNS float
 - Retorna a maior velocidade atingida do *trajectory* passado como parâmetro, em quilômetros por hora.
- CREATE FUNCTION low_speed(trajectory trajectory) RETURNS float
 - Retorna a menor velocidade atingida do *trajectory* passado como parâmetro, em quilômetros por hora.
- CREATE FUNCTION positions(trajectory trajectory) RETURNS SETOF position
 - Retorna um conjunto composto por todos os *positions* do *trajectory* passado como parâmetro.

3.2.2) Extensão do PostGIS

O PostGIS é uma extensão do PostgreSQL que adiciona suporte a objetos geográficos [10]. Há diversas funções de relacionamento topológicos implementadas e otimizadas no PostGIS. Uma forma de tornar possível utilizar todas essas funções com os novos tipos de dado é realizando a conversão para um tipo nativo do PostGIS, i.e. realizar a conversão para os tipos *Geometry* e *Geography*. Enquanto que o tipo *Geometry* trata de medidas cartesianas, o tipo *Geography* trata de medidas geodésicas [10], sendo o segundo ideal para fazer uso de valores de latitude e longitude.

Porém, os tipos de dados que o PostGIS trata são puramente espaciais, diferente dos tipos propostos, que possuem também informação temporal. Dessa forma, não é possível realizar a conversão inversa de forma trivial, i.e. obter dados de *Position* ou *Trajectory* a partir de dados do tipo *Geometry* ou *Geography*.

Já existe implementado no PostGIS funções de conversão para os tipos geométricos nativos do PostgreSQL. Logo, a implementação das funções de conversão para os novos tipos *Position* e *Trajectory* foi feita seguindo as funções de conversão já existentes. O código presente da Listagem 3.7 representa a criação das novas funções:

```
01. CREATE OR REPLACE FUNCTION geometry(position)
02. RETURNS geometry
03. AS '$libdir/postgis-2.1','position_to_geometry'
04. LANGUAGE 'c' IMMUTABLE STRICT;
05.
06. CREATE OR REPLACE FUNCTION geometry(trajjectory)
07. RETURNS geometry
08. AS '$libdir/postgis-2.1','trajectory_to_geometry'
09. LANGUAGE 'c' IMMUTABLE STRICT;
```

Listagem 3.7 – Criação das funções de conversão no PostGIS.

Para utilizar o tipo *Geography* é possível realizar um cast conforme mostra a Listagem 3.8:

```
01. SELECT geography(geometry(trajjectory));
```

Listagem 3.8 – Exemplo de cast para o tipo *geography*.

A partir da utilização das funções de conversão, é possível utilizar todas as funções implementadas no PostGIS para os tipos *Geometry* e *Geography*. Por exemplo, as funções para o tipo de dado *Trajectory* que envolvem *length* foram construídas fazendo uso da função *ST_Length*, já definida no PostGIS para os tipos *Geometry* e *Geography*. A Listagem 3.9 mostra como é possível utilizar esta função com o tipo proposto, tanto fazendo a conversão para o tipo *Geometry* quanto para o tipo *Geography*.

```
01.      SELECT St_Length(geometry(trajjectory))
02.      FROM tabela_exemplo;
03.
04.      SELECT St_Length(geography(geometry(trajjectory)))
05.      FROM tabela_exemplo;
```

Listagem 3.9 – Uso da função *ST_Length*.

3.2.3) Adicionando Semântica ao tipo *Trajectory*

Conforme exposto na Seção 2.2.4, existem diversos métodos capazes de extrair elementos que podem representar *stops* e *moves* de objetos móveis. A fim de demonstrar que é possível aplicar os métodos descritos no TAD *Trajectory*, foi implementado um deles, o CB-SMoT. A Listagem 3.10 apresenta a assinatura da função proposta.

```
06.      CREATE FUNCTION cb_smot(trajjectory trajjectory, threshold
      float, minduration int) RETURNS SETOF trajjectory $$
```

Listagem 3.10 – Assinatura da função *cb_smot*.

O método CB-SMoT é capaz de encontrar candidatos à *stops* com base na variação de velocidade. A função criada recebe como parâmetro: *trajjectory* do tipo *Trajectory*, *threshold* do tipo *float* e *minduration* do tipo *integer*. Conforme descrito na Seção 2.2.4, um trecho da trajetória pode ser considerado um candidato à *stop* se a velocidade está abaixo de um determinado limiar (*threshold*) durante um intervalo de tempo mínimo (*minduration*). Está sendo utilizado velocidade em quilômetros por hora (km/h) e duração em segundos. A função retorna um conjunto de elementos do tipo *Trajectory*, que pode ser interpretado como sendo os candidatos a *stop* encontrados na trajetória de acordo com os parâmetros utilizados.

O passo seguinte para encontrar de fato os *stops*, consiste em realizar sobreposições com feições dos locais de interesse relativos ao domínio da aplicação. Isso pode ser feito utilizando funções do PostGIS como a função `ST_Intersects`, `ST_Within`/`ST_Contains`, `ST_DWithin`, que são descritas como segue:

- `ST_Intersects(geometry A, geometry B)`
 - Retorna *true* se existe pelo menos um ponto em comum entre as geometrias.
- `ST_Within`/`ST_Contains`(geometry A, geometry B)
 - Retorna *true* se a geometria A está completamente dentro da geometria B.
- `ST_DWithin`(geometry A, geometry B, radius)
 - Retorna *true* se as geometrias estão dentro da distância especificada (*radius*) um do outro. Ideal para casos onde não se tem a feição do local, apenas um ponto. Pode ser utilizado pontos como sendo os locais de interesse e determinar um valor para ser utilizado como raio de distância para cada ponto.

3.3) Considerações Finais

Nesse capítulo, foi apresentada uma proposta de implementação do tipo de dados *Trajectory* para o PostgreSQL/PostGIS, incluindo a definição de funções para criação, instanciação e manipulação do mesmo. Algumas destas funções de manipulação permitem a obtenção de informações semânticas de trajetórias. No capítulo seguinte, é detalhada uma aplicação baseada em dados reais de trajetórias para validação do tipo proposto neste capítulo.

4) Capítulo 4: Validação do TAD Proposto

4.1) Introdução

Este capítulo tem como objeto mostrar a utilização do tipo de dado proposto e desenvolvido durante esse trabalho. Na Seção 4.2, será apresentada uma aplicação utilizando o TAD *Trajectory*, mostrando as funcionalidades que foram desenvolvidas para o armazenamento e a manipulação do mesmo. Em seguida, a Seção 4.3 trará as conclusões deste capítulo.

4.2) Aplicação utilizando o TAD proposto

A partir de dados reais de trajetórias brutas coletados através de dispositivos GPS em 10.357 taxis durante o período de 2 a 8 de fevereiro de 2008 em Beijing, China [5] [6] foi possível construir uma base de dados estruturada de acordo com o esquema conceitual da Figura 4.1:



Figura 4.1 – Modelo Entidade-Relacionamento para a aplicação desenvolvida

Os dados estão dispostos em arquivo texto e são formatados de acordo com o conteúdo da Tabela 4.1.

taxi id, date time, longitude, latitude
1,2008-02-02 15:36:08,116.51172,39.92123
1,2008-02-02 15:46:08,116.51135,39.93883
1,2008-02-02 15:56:08,116.51135,39.93883
1,2008-02-02 16:06:08,116.51627,39.91034

Tabela 4.1 – Amostra dos dados utilizados

A média de intervalo de todas as amostras é de 177 segundos com uma distância de cerca de 623 metros [5] [6]. A partir de uma parcela de amostras, convertidas em comandos *INSERT INTO*, foi possível realizar a inserção dos dados na tabela *Percurso*.

Primeiramente, foi necessário realizar a criação das tabelas. Deve ser possível criar uma nova tabela utilizando o TAD *Trajectory* assim como ocorre com qualquer outro tipo presente no PostgreSQL. O código de criação das tabelas baseia-se no esquema entidade relacionamento da Figura 4.1, e é apresentado na Listagem 4.1.

```
01. CREATE TABLE taxi(  
02.     id BIGSERIAL NOT NULL,  
03.     PRIMARY KEY ("id")  
04. );  
05.  
06. CREATE TABLE percurso(  
07.     id BIGSERIAL NOT NULL,  
08.     taxi_id BIGINT NOT NULL,  
09.     trajetoria trajectory NOT NULL,  
10.     FOREIGN KEY("taxi_id") REFERENCES taxi("id"),  
11.     PRIMARY KEY ("id")  
12. );
```

Listagem 4.1 – Script de criação das tabelas da aplicação.

O passo seguinte consistiu em armazenar dados nas tabelas criadas. Deve ser possível armazenar dados referentes ao TAD *Trajectory*, conforme descrito na Seção 3.2.1. Comandos *INSERT INTO* foram gerados a partir de uma parcela dos dados e então inseridos com sucesso.

Uma vez criadas, as tabelas, e tendo os dados inseridos, foi possível fazer uso das funções definidas na Seção 3.2.1 para responder às questões que seguem:

- Quantos quilômetros foram percorridos pelo Táxi de Id 8179?

```
01. SELECT SUM(length(trajetoria))/1000  
02. FROM percurso WHERE taxi_id = 8179  
03. R: 9.90287685350241
```

- Qual táxi trabalhou por mais tempo e quanto foi o tempo em horas?

```
01. SELECT  
02.     taxi_id,  
03.     SUM(duration(trajetoria))/360 as tempo_total  
04. FROM  
05.     percurso  
06. GROUP BY taxi_id  
07. ORDER BY tempo_total DESC  
08. LIMIT 1;  
09.  
10. R: 8179;4.958333333333333
```


- Qual o tamanho do menor percurso realizado (em quilômetros)?

```
01. SELECT MIN(length(trajetoria))/1000
02. FROM percurso
03. R: 0.963336647871536
```

- Quantos quilômetros foram percorridos ao total por todos os táxis?

```
01. SELECT SUM(length(trajetoria))/1000
02. FROM percurso
03. R: 21.0578074981487
```

- Qual a média de tempo em segundos entre a captura dos *Positions*?

```
01. SELECT AVG(avg_duration(trajetoria))
02. FROM percurso
03. R: 2.27337278106509
```

- Qual a velocidade máxima atingida (km/h) entre todos os percursos?

```
01. SELECT MAX(top_speed(trajetoria))
02. FROM percurso
03. R: 102.093629555018
```

- Quais táxis estavam trabalhando entre X e Y?

```
01. SELECT taxi_id FROM percurso
02. WHERE segment(trajetoria, '2008-02-03
03. 08:00'::timestamp, '2008-02-03 12:00'::timestamp)
04. is not null
R: 3557
```

- Qual táxi passou mais tempo parado e quanto foi esse tempo em minutos?

```
01. SELECT
02. taxi_id, SUM(duration(S.stop))/60 as tempo_parado
03. FROM
04. (SELECT cb_smot(trajetoria, 0, 60) as stop, taxi_id
05. FROM percurso) S
06. GROUP BY S.taxi_id
07. ORDER BY tempo_parado DESC
08. LIMIT 1;
09. R: 6275;2.78333333333333
```

Observação: Para essa questão foi utilizado o método CB-SMoT considerando paradas de duração maior do que 60 segundos com velocidade igual a zero (parado de fato).

4.3) Conclusão

A finalidade deste capítulo era mostrar que é possível utilizar o TAD *Trajectory* como qualquer outro tipo de dado já definido no PostgreSQL. Um dos principais objetos da criação desse novo tipo é facilitar a consulta, inserção, remoção e análise de dados de trajetórias.

Por meio da criação de uma aplicação que utiliza o tipo de dado proposto e as suas funções definidas, foi possível manipular dados reais de trajetórias, demonstrando que o novo tipo de dado pode ser utilizado como qualquer outro tipo definido no SGBD e foi ainda possível, ilustrar como são obtidas informações semânticas a partir do uso das funções descritas na Seção 3.2.1, atingindo assim o objetivo deste capítulo.

.

5) Capítulo 5: Conclusão

5.1) Introdução

Este capítulo apresenta as conclusões sobre o trabalho realizado. A Seção 5.2 descreve suas principais contribuições. Por fim, a Seção 5.3 indica possíveis trabalhos futuros envolvendo o tipo de dado proposto.

5.2) Principais Contribuições

Este trabalho propôs a especificação e implementação de um novo tipo abstrato de dados no SGBD PostgreSQL, chamado *Trajectory*, de maneira que o tipo de dado proposto possa ser utilizado como qualquer outro tipo existente no SGBD. Além da especificação e implementação, foi realizada uma validação através do desenvolvimento de uma aplicação que utiliza o novo tipo proposto.

Nesse sentido, este trabalho contribui com a proposta do TAD *Trajectory* como alternativa para o armazenamento e manipulação de dados de trajetória, com a finalidade de facilitar a consulta, inserção, remoção e análise de dados desse tipo.

5.3) Trabalhos Futuros

Os seguintes pontos podem ser abordados como trabalhos futuros desta pesquisa:

- Investigar extensões na linguagem SQL para incorporar operadores e funções específicos para o tipo de dados proposto.
- Realizar avaliações de desempenho com o tipo de dado *Trajectory* e investigar ganhos e perdas com relação ao uso de tipos básicos existentes.
- Desenvolver aplicações de Data Warehouse de trajetórias com base no tipo implementado e numa arquitetura ROLAP.
- Estender a especificação do TAD proposto para permitir a representação de trajetórias de grupos de objetos móveis (moving regions).
- Estender a especificação do TAD proposto para permitir a representação de outros tipos de trajetórias: dados tridimensionais; dados espaciais do tipo *raster* ou trajetórias coletadas no formato de big data.

Referências Bibliográficas

- [1] Alvares, L. O., Bogorny, V., Kuijpers, B., de Macedo, J. A. F., Moelans, B., and Vaisman, A. (2007). A model for enriching trajectories with semantic geographical information. In *Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, GIS '07, pages 22:1–22:8, New York, NY, USA. ACM.
- [2] Dictionary of Algorithms and Data Structures (N. do T.). Disponível em < <http://xlinux.nist.gov/dads/>>. Acesso em Fevereiro de 2014.
- [3] FOLDOC - Free On-Line Dictionary of Computing (N. do T.). Disponível em: <http://foldoc.org/>>. Acesso em Fevereiro de 2014.
- [4] Güting, R. H., Behr, T., Almeida, V., Ding, Z., Hoffmann, F., Spiekermann, M., ... Hagen, D.-. (n.d.). SECONDO: An Extensible DBMS Architecture and Prototype *.
- [5] Jing Yuan, Yu Zheng, Xing Xie, and Guangzhong Sun. Driving with knowledge from the physical world. In *The 17th ACM SIGKDD international conference on Knowledge Discovery and Data mining*, KDD '11, New York, NY, USA, 2011. ACM.
- [6] Jing Yuan, Yu Zheng, Chengyang Zhang, Wenlei Xie, Xing Xie, Guangzhong Sun, and Yan Huang. T-drive: driving directions based on taxi trajectories. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '10, pages 99{108, New York, NY, USA, 2010. ACM.
- [7] Kisilevich, S., Mansmann, F., Nanni, M., & Rinzivillo, S. (2010). Spatio-Temporal Clustering: a Survey. *Data Mining and Knowledge Discovery Handbook*, 1–22. doi:10.1007/978-0-387-09823-4
- [8] Palma, A. T., Bogorny, V., Kuijpers, B., and Alvares, L. O. (2008). A clustering-based approach for discovering interesting places in trajectories. In *Proceedings of the 2008 ACM symposium on Applied Computing*, SAC'08, (December), 863.
- [9] Pelekis, N., Frentzos, E., Giatrakos, N., & Theodoridis, Y. (n.d.). HERMES: Aggregative LBS via a Trajectory DB Engine, 9–11.

[10] POSTGIS, home. Disponível em: <<http://postgis.refractory.net/>>. Acesso em: Novembro de 2013.

[11] POSTGRESQL, documentation. Disponível em: <<http://www.postgresql.org/docs/9.2/static/docguide.html>>. Acesso em: Novembro de 2013.

[12] Rocha, J. A. M. R., Times, V. C., Oliveira, G., Alvares, L. O., and Bogorny, V. (2010). Db-smot: A direction-based spatio-temporal clustering method. In *IEEE Conf. of Intelligent Systems*, pages 114–119. IEEE.

[13] Spaccapietra, S., Parent, C., Damiani, M. L., de Macedo, J. A., Porto, F., and Vangenot, C. (2008). A conceptual view on trajectories. *Data Knowl. Eng.*, 65(1), 126–146.

Apêndice A - Scripts na linguagem C

position.h

```
#ifndef POSITION_H
#define POSITION_H

#include "postgres.h"
#include <libpq/pqcomm.h>
#include <utils/timestamp.h>
#include <utils/geo_decls.h>

typedef struct Position{
    double x;
    double y;
    Timestamp t;
} Position;

extern Datum position_in(PG_FUNCTION_ARGS);
extern Datum make_position(PG_FUNCTION_ARGS);
extern Datum position_out(PG_FUNCTION_ARGS);
extern Datum position_add(PG_FUNCTION_ARGS);
extern Datum position_sub(PG_FUNCTION_ARGS);
extern Datum to_point(PG_FUNCTION_ARGS);
extern Datum get_timestamp(PG_FUNCTION_ARGS);

#endif
```

trajectory.h

```
#ifndef TRAJECTORY_H
#define TRAJECTORY_H

#include "postgres.h"
#include "fmgr.h"
#include <utils/timestamp.h>
#include <utils/geo_decls.h>
#include "position.h"
#include <string.h>

typedef struct Trajectory{
    int32 vl_len_;
    int32 nposs;
    Position p[1];
} Trajectory;

extern Datum trajectory_in(PG_FUNCTION_ARGS);
extern Datum trajectory_out(PG_FUNCTION_ARGS);
extern Datum trajectory_nposs(PG_FUNCTION_ARGS);
extern Datum trajectory_head(PG_FUNCTION_ARGS);
extern Datum trajectory_tail(PG_FUNCTION_ARGS);
extern Datum trajectory_n(PG_FUNCTION_ARGS);
extern Datum trajectory_segment(PG_FUNCTION_ARGS);
extern Datum trajectory_segment_interval(PG_FUNCTION_ARGS);
extern Datum trajectory_add(PG_FUNCTION_ARGS);

#endif
```

trajectory.c

```
#include "position.h"
#include "trajectory.h"

//#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
//endif

/*
 * Delimiters for input and output strings.
 * LDELIM, RDELIM, and DELIM are left, right, and separator delimiters,
 * respectively.
 * LDELIM_EP, RDELIM_EP are left and right delimiters for paths with
 * endpoints.
 */

#define LDELIM '('
#define RDELIM ')'
#define DELIM ','
#define LDELIM_EP '['
#define RDELIM_EP ']'

/*

    Position
    Author: Paulo Carvalho (penc@cin.ufpe.br)

 */

PG_FUNCTION_INFO_V1(position_in);

Datum
position_in(PG_FUNCTION_ARGS)
{
    char *str = PG_GETARG_CSTRING(0);
    double x;
    double y;
    char *date = (char *) palloc(32);
    char *time = (char *) palloc(32);
    char *timestamp = (char *) palloc(64);
    Timestamp t;

    while (isspace((unsigned char) *str))
        str++;

    if (sscanf(str, "(%lf,%lf,%s%s)", &x, &y, date, time) == 4){
        snprintf(timestamp, 64, "%s %s", date, time);
    }else if (sscanf(str, "(%lf,%lf,%s)", &x, &y, date) == 3){
        timestamp = date;
    }else{
        ereport(ERROR, (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
        errmsg("invalid input syntax for position: \"%s\"", str)));
        PG_RETURN_NULL();
    }
    Position *result = (Position *) palloc(sizeof(Position));
    result->x = x;
    result->y = y;
    t = DatumGetTimestamp(DirectFunctionCall3(timestamp_in,
    CStringGetDatum(timestamp), 0, -1));
```

```

        result->t = t;
        PG_RETURN_POINTER(result);
    }

PG_FUNCTION_INFO_V1(make_position);

Datum
make_position(PG_FUNCTION_ARGS)
{
    if (PG_ARGISNULL(0) || PG_ARGISNULL(1))
        PG_RETURN_NULL();

    Point *p = (Point *) PG_GETARG_POINTER(0);
    Timestamp t = PG_GETARG_TIMESTAMP(1);
    Position *result = (Position *) malloc(sizeof(Position));
    result->x = p->x;
    result->y = p->y;
    result->t = t;
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(position_out);

Datum
position_out(PG_FUNCTION_ARGS)
{
    if (PG_ARGISNULL(0))
        PG_RETURN_NULL();

    Position *position = (Position *) PG_GETARG_POINTER(0);

    if (!position){
        PG_RETURN_NULL();
    }else{
        int size = (sizeof(char) *128);
        char *result = malloc(size);
        char *str = DatumGetCString(DirectFunctionCall1(timestamp_out,
position->t));
        snprintf(result, size, "(%.11g,%.11g,%s)", position->x, position-
>y, str);
        PG_RETURN_CSTRING(result);
    }
}

PG_FUNCTION_INFO_V1(position_add);

Datum
position_add(PG_FUNCTION_ARGS)
{
    if (PG_ARGISNULL(0) || PG_ARGISNULL(1))
        PG_RETURN_NULL();

    Position *a = (Position *) PG_GETARG_POINTER(0);
    Position *b = (Position *) PG_GETARG_POINTER(1);

    if (!a || !b)
        PG_RETURN_NULL();

    Position *result = (Position *) malloc(sizeof(Position));
    result->x = (a->x + b->x);
    result->y = (a->y + b->y);

```



```

        result->t = ((a->t > b->t) ? a->t : b->t);
        PG_RETURN_POINTER(result);
    }

PG_FUNCTION_INFO_V1(position_sub);

Datum
position_sub(PG_FUNCTION_ARGS)
{
    if (PG_ARGISNULL(0) || PG_ARGISNULL(1))
        PG_RETURN_NULL();

    Position *a = (Position *) PG_GETARG_POINTER(0);
    Position *b = (Position *) PG_GETARG_POINTER(1);

    if (!a || !b)
        PG_RETURN_NULL();

    Position *result = (Position *) calloc(sizeof(Position));
    result->x = (a->x - b->x);
    result->y = (a->y - b->y);
    result->t = ((a->t < b->t) ? a->t : b->t);
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(to_point);

Datum
to_point(PG_FUNCTION_ARGS)
{
    if (PG_ARGISNULL(0))
        PG_RETURN_NULL();

    Position *position = (Position *) PG_GETARG_POINTER(0);

    if (!position)
        PG_RETURN_NULL();

    Point *result = (Point *) calloc(sizeof(Point));
    char *str = (char *) calloc(100);
    snprintf(str, 100, "(%g, %g)", position->x, position->y);
    result = (Point *) DatumGetPointer(DirectFunctionCall1(point_in,
CStringGetDatum(str)));
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(get_timestamp);

Datum
get_timestamp(PG_FUNCTION_ARGS)
{
    if (PG_ARGISNULL(0))
        PG_RETURN_NULL();

    Position *position = (Position *) PG_GETARG_POINTER(0);

    if (!position)
        PG_RETURN_NULL();

    PG_RETURN_TIMESTAMP(position->t);
}

```

```

/*
    Trajectory
    Author: Paulo Carvalho (penc@cin.ufpe.br)
*/

static int trio_count(char *s, char delim);

static int
trio_count(char *s, char delim)
{
    int ndelim = 0;

    while ((s = strchr(s, delim)) != NULL){
        ndelim++;
        s++;
    }
    if (ndelim == 0)
        return 0;
    else
        return (ndelim % 3) ? ((ndelim + 1) / 3) : -1;
}

PG_FUNCTION_INFO_V1(trajjectory_in);

Datum
trajjectory_in(PG_FUNCTION_ARGS)
{
    char *str = PG_GETARG_CSTRING(0);

    if (str != NULL){
        Trajectory *trajectory;
        char *s;
        int nposs;
        int size;
        int base_size;
        int depth = 0;

        if ((nposs = trio_count(str, ',')) < 0)
            ereport(ERROR,
                    (errmsg("invalid input syntax for type trajectory: \"%s\"",
str)));

        s = str;
        while (isspace((unsigned char) *s))
            s++;

        /*skip single leading paren */
        if ((*s == LDELIM_EP) && (strchr(s, LDELIM_EP) == s)){
            s++;
            depth++;
        }

        base_size = sizeof(trajectory->p[0]) * nposs;
        size = offsetof(Trajectory, p[0]) + base_size;

        /*Check for integer overflow */

```

```

        if ((npos > 0 && base_size / npos != sizeof(traj->p[0])) ||
size <= base_size)
            ereport(ERROR,
                (errcode(ERRCODE_PROGRAM_LIMIT_EXCEEDED),
                 errmsg("too many positions requested")));

        trajectory = (Trajectory *) palloc(size);

        SET_VARSIZE(trajectory, size);
        trajectory->npos = npos;

        int i = 0;
        char *token;
        Position *p;
        Position *tmp;
        Position *preceding;

        while((token = strsep(&s, " ")) != NULL && token[0] != '\0' &&
*token != RDELIM_EP){
            p = &(trajectory->p[i]);
            tmp = (Position *)
DatumGetPointer(DirectFunctionCall1(position_in, CStringGetDatum(token)));

            if (i > 0){
                preceding = &(trajectory->p[i-1]);
                if (preceding->t >= tmp->t)
                    ereport(ERROR,
                        (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                         errmsg("the positions have to be ordered by time ASC...
has at least one out of order.")));
            }

            p->x = tmp->x;
            p->y = tmp->y;
            p->t = tmp->t;
            i++;

            if(*s == DELIM)
                s++;
        }

        PG_RETURN_POINTER(trajectory);
    }else{
        PG_RETURN_NULL();
    }
}

PG_FUNCTION_INFO_V1(trajectory_out);

Datum
trajectory_out(PG_FUNCTION_ARGS)
{
    if (PG_ARGISNULL(0))
        PG_RETURN_NULL();

    Trajectory *trajectory = (Trajectory *) PG_GETARG_POINTER(0);

    if (!trajectory){
        PG_RETURN_NULL();
    }else{

```

```

    int i;
    int size = (sizeof(char) * (4 + (128 * trajectory->nposs)));
    char *result = (char *) palloc(size);
    if (trajectory->nposs > 0){
        char *temp = (char *) palloc(size);
        result[0] = '[';
        for (i = 0; i < trajectory->nposs; i++){
            if (i == 0)
                snprintf(result, size, "[%s",
DatumGetCString(DirectFunctionCall1(position_out,
PointerGetDatum(&trajectory->p[i]))));
            else
                snprintf(result, size, "%s,%s", temp,
DatumGetCString(DirectFunctionCall1(position_out,
PointerGetDatum(&trajectory->p[i]))));
            strcpy(temp, result);
        }
        strcpy(temp, result);
        snprintf(result, size, "%s%s", temp, "]");
    }else{
        snprintf(result, size, "[]");
    }
    PG_RETURN_CSTRING(result);
}

PG_FUNCTION_INFO_V1(trajectory_nposs);

Datum
trajectory_nposs(PG_FUNCTION_ARGS)
{
    if (PG_ARGISNULL(0))
        PG_RETURN_INT32(0);

    Trajectory *trajectory = (Trajectory *) PG_GETARG_POINTER(0);

    if (!trajectory){
        PG_RETURN_INT32(0);
    }else{
        PG_RETURN_INT32(trajectory->nposs);
    }
}

PG_FUNCTION_INFO_V1(trajectory_head);

Datum
trajectory_head(PG_FUNCTION_ARGS)
{
    if (PG_ARGISNULL(0))
        PG_RETURN_NULL();

    Trajectory *trajectory = (Trajectory *) PG_GETARG_POINTER(0);

    if (!trajectory){
        PG_RETURN_NULL();
    }else{
        if (trajectory->nposs > 0)
            PG_RETURN_POINTER(&(trajectory->p[0]));
        else
            PG_RETURN_NULL();
    }
}

```

```

}

PG_FUNCTION_INFO_V1(trajjectory_tail);

Datum
trajjectory_tail(PG_FUNCTION_ARGS)
{
    if (PG_ARGISNULL(0))
        PG_RETURN_NULL();

    Trajectory *trajjectory = (Trajectory *) PG_GETARG_POINTER(0);

    if (!trajjectory){
        PG_RETURN_NULL();
    }else{
        int tail = trajjectory->nposs - 1;
        if (tail >= 0)
            PG_RETURN_POINTER(&(trajjectory->p[tail]));
        else
            PG_RETURN_NULL();
    }
}

PG_FUNCTION_INFO_V1(trajjectory_n);

Datum
trajjectory_n(PG_FUNCTION_ARGS)
{
    if (PG_ARGISNULL(0) || PG_ARGISNULL(1))
        PG_RETURN_NULL();

    Trajectory *trajjectory = (Trajectory *) PG_GETARG_POINTER(0);
    int n = PG_GETARG_INT32(1);

    if (!trajjectory || n < 0 || n >= trajjectory->nposs){
        PG_RETURN_NULL();
    }else{
        PG_RETURN_POINTER(&(trajjectory->p[n]));
    }
}

PG_FUNCTION_INFO_V1(trajjectory_add);

Datum
trajjectory_add(PG_FUNCTION_ARGS)
{
    if (PG_ARGISNULL(0) || PG_ARGISNULL(1))
        PG_RETURN_NULL();

    Trajectory *t = (Trajectory *) PG_GETARG_POINTER(0);
    Position *position = (Position *) PG_GETARG_POINTER(1);

    if (!t)
        PG_RETURN_NULL();
    else if (!position)
        PG_RETURN_POINTER(t);

    Position *tail = (Position *)
DatumGetPointer(DirectFunctionCall1(trajjectory_tail, PointerGetDatum(t)));

    if(position->t > tail->t){

```

```

Trajectory *trajectory;
int nposs = t->nposs + 1;
int base_size = sizeof(trajecory->p[0]) *nposs;
int size = offsetof(Trajectory, p[0]) + base_size;
trajectory = (Trajectory *) palloc(size);
SET_VARSIZE(trajectory, size);
trajectory->nposs = nposs;

int i = 0;
char *token;
Position *p;
Position *tmp;

while(i < t->nposs){
    p = &(trajectory->p[i]);
    tmp = &(t->p[i]);
    p->x = tmp->x;
    p->y = tmp->y;
    p->t = tmp->t;
    i++;
}
p = &(trajectory->p[i]);
p->x = position->x;
p->y = position->y;
p->t = position->t;
PG_RETURN_POINTER(trajectory);
}else{
    ereport(ERROR,
        (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION) ,
        errmsg("invalid operation. the position cannot be add on trajectory
because of your timestamp.")));
}
}

PG_FUNCTION_INFO_V1(trajectory_segment);

Datum
trajectory_segment(PG_FUNCTION_ARGS)
{
    if (PG_ARGISNULL(0) || PG_ARGISNULL(1) || PG_ARGISNULL(2))
        PG_RETURN_NULL();

    Trajectory *t = (Trajectory *) PG_GETARG_POINTER(0);
    int s = PG_GETARG_INT32(1);
    int e = PG_GETARG_INT32(2);

    if (!t)
        PG_RETURN_NULL();

    if (s < 0 || s > e || e > (t->nposs - 1))
        ereport(ERROR,
            (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION) ,
            errmsg("invalid segment. \"%d, %d\"", s, e)));
    else if (t == NULL)
        PG_RETURN_NULL();

    Trajectory *trajectory;
    int nposs = 1 + e - s;
    int base_size = sizeof(trajecory->p[0]) *nposs;
    int size = offsetof(Trajectory, p[0]) + base_size;
    trajectory = (Trajectory *) palloc(size);

```

```

SET_VARSIZE(trajjectory, size);
trajjectory->nposs = nposs;

int i = 0;
char *token;
Position *p;
Position *tmp;

while(s <= e){
    p = &(trajjectory->p[i]);
    tmp = &(t->p[s]);
    p->x = tmp->x;
    p->y = tmp->y;
    p->t = tmp->t;
    i++;s++;
}
PG_RETURN_POINTER(trajjectory);
}

PG_FUNCTION_INFO_V1(trajjectory_segment_interval);

Datum
trajjectory_segment_interval(PG_FUNCTION_ARGS)
{
    if (PG_ARGISNULL(0) || PG_ARGISNULL(1) || PG_ARGISNULL(2))
        PG_RETURN_NULL();

    Trajectory *t = (Trajectory *) PG_GETARG_POINTER(0);
    Timestamp s = PG_GETARG_TIMESTAMP(1);
    Timestamp e = PG_GETARG_TIMESTAMP(2);

    if (!t)
        PG_RETURN_NULL();

    Position *tail = (Position *)
DatumGetPointer(DirectFunctionCall1(trajjectory_tail, PointerGetDatum(t)));
    Position *head = (Position *)
DatumGetPointer(DirectFunctionCall1(trajjectory_head, PointerGetDatum(t)));

    if (s > e)
        ereport(ERROR,
            (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
              errmsg("invalid segment interval.")));
    else if (s > tail->t || e < head->t || !t)
        PG_RETURN_NULL();

    int start = -1;
    int end = t->nposs - 1;
    int i = 0;
    Position *p;

    while(i < t->nposs){
        p = &(t->p[i]);

        if(p->t >= s && start == -1)
            start = i;

        if(p->t > e){
            end = i - 1;
            break;
        }
    }

```

```

        i++;
    }
    Trajectory *trajectory = (Trajectory *)
DatumGetPointer(DirectFunctionCall3(trajectory_segment, PointerGetDatum(t),
Int32GetDatum(start), Int32GetDatum(end)));
    PG_RETURN_POINTER(trajectory);
}

Datum
trajectory_segment_box(PG_FUNCTION_ARGS)
{
    if (PG_ARGISNULL(0) || PG_ARGISNULL(1) || PG_ARGISNULL(2))
        PG_RETURN_NULL();

    Trajectory *t = (Trajectory *) PG_GETARG_POINTER(0);
    Point s = PG_GETARG_TIMESTAMP(1);
    Point e = PG_GETARG_TIMESTAMP(2);

    if (!t)
        PG_RETURN_NULL();

    Position *tail = (Position *)
DatumGetPointer(DirectFunctionCall1(trajectory_tail, PointerGetDatum(t)));
    Position *head = (Position *)
DatumGetPointer(DirectFunctionCall1(trajectory_head, PointerGetDatum(t)));

    if (s > e)
    {
        ereport(ERROR,
            (errmsg("invalid segment interval.")));
    }
    else if (s > tail->t || e < head->t || !t)
        PG_RETURN_NULL();

    int start = -1;
    int end = t->nposs - 1;
    int i = 0;
    Position *p;

    while(i < t->nposs){
        p = &(t->p[i]);

        if(p->t >= s && start == -1)
            start = i;

        if(p->t > e){
            end = i - 1;
            break;
        }
        i++;
    }

    Trajectory *trajectory = (Trajectory *)
DatumGetPointer(DirectFunctionCall3(trajectory_segment, PointerGetDatum(t),
Int32GetDatum(start), Int32GetDatum(end)));
    PG_RETURN_POINTER(trajectory);
}

```

Extensão do arquivo: ***geometry_inout.c***


```

Datum position_to_geometry(PG_FUNCTION_ARGS);
Datum trajectory_to_geometry(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(position_to_geometry);
Datum position_to_geometry(PG_FUNCTION_ARGS)
{
    Position *position;
    LWPOINT *lwpoint;
    GSERIALIZED *geom;

    POSTGIS_DEBUG(2, "position_to_geometry called");

    if ( PG_ARGISNULL(0) )
        PG_RETURN_NULL();

    position = (Position *) PG_GETARG_POINTER(0);

    if ( ! position )
        PG_RETURN_NULL();

    lwpoint = lwpoint_make2d(SRID_UNKNOWN, position->x, position->y);
    geom = geometry_serialize(lwpoint_as_lwgeom(lwpoint));
    lwpoint_free(lwpoint);
    PG_RETURN_POINTER(geom);
}

PG_FUNCTION_INFO_V1(trajectory_to_geometry);
Datum trajectory_to_geometry(PG_FUNCTION_ARGS)
{
    Trajectory *trajectory;
    LWLINE *lwline;
    POINTARRAY *pa;
    GSERIALIZED *geom;
    POINT4D pt;
    Position p;
    int i;
    POSTGIS_DEBUG(2, "trajectory_to_geometry called");

    if ( PG_ARGISNULL(0) )
        PG_RETURN_NULL();

    trajectory = (Trajectory *) PG_GETARG_POINTER(0);

    if ( ! trajectory )
        PG_RETURN_NULL();

    pa = ptarray_construct_empty(0, 0, trajectory->nposs);

    for ( i = 0; i < trajectory->nposs; i++ )
    {
        p = trajectory->p[i];
        pt.x = p.x;
        pt.y = p.y;
        ptarray_append_point(pa, &pt, LW_FALSE);
    }
    lwline = lwline_construct(SRID_UNKNOWN, NULL, pa);
    geom = geometry_serialize(lwline_as_lwgeom(lwline));
    lwline_free(lwline);
    PG_RETURN_POINTER(geom);
}

```

Apêndice B - Scripts na linguagem PL/pgSQL

trajectory.sql

```
DROP TYPE "position" CASCADE;
DROP TYPE trajectory CASCADE;

--Position

CREATE type "position";

CREATE function position_in(cstring) RETURNS "position" AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so', 'position_in'
LANGUAGE C IMMUTABLE STRICT;

CREATE function position_out("position") RETURNS cstring AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so', 'position_out'
LANGUAGE C IMMUTABLE STRICT;

CREATE function to_point("position") RETURNS point AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so', 'to_point'
LANGUAGE C IMMUTABLE STRICT;

CREATE function get_timestamp("position") RETURNS timestamp AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so', 'get_timestamp'
LANGUAGE C IMMUTABLE STRICT;

CREATE function make_position(point, timestamp) RETURNS "position" AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so', 'make_position'
LANGUAGE C IMMUTABLE STRICT;

CREATE function make_position(point, timestamp with time zone) RETURNS
"position" AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so', 'make_position'
LANGUAGE C IMMUTABLE STRICT;

CREATE function position_add("position", "position") RETURNS "position" AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so', 'position_add'
LANGUAGE C IMMUTABLE STRICT;

CREATE function position_sub("position", "position") RETURNS "position" AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so', 'position_sub'
LANGUAGE C IMMUTABLE STRICT;

CREATE type "position"(
    internallength = 24,
    input = position_in,
    output = position_out,
    storage = plain
);

CREATE OPERATOR +(
    leftarg = "position",
    rightarg = "position",
    procedure = position_add,
    commutator = +
);

CREATE OPERATOR -(
```

```

    leftarg = "position",
    rightarg = "position",
    procedure = position_sub,
    commutator = -
);

--Trajectory

CREATE type trajectory;

CREATE function trajectory_in(cstring) RETURNS trajectory AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so', 'trajectory_in'
LANGUAGE C IMMUTABLE STRICT;

CREATE function trajectory_out(trajectory) RETURNS cstring AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so', 'trajectory_out'
LANGUAGE C IMMUTABLE STRICT;

CREATE type trajectory(
    internallength = VARIABLE,
    input = trajectory_in,
    output = trajectory_out,
    storage = main
);

CREATE function npositions(trajectory) RETURNS int AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so', 'trajectory_nposs'
LANGUAGE C IMMUTABLE STRICT;

CREATE function trajectory_start(trajectory) RETURNS "position" AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so', 'trajectory_head'
LANGUAGE C IMMUTABLE STRICT;

CREATE function trajectory_end(trajectory) RETURNS "position" AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so', 'trajectory_tail'
LANGUAGE C IMMUTABLE STRICT;

CREATE function position_n(trajectory, int) RETURNS "position" AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so', 'trajectory_n'
LANGUAGE C IMMUTABLE STRICT;

CREATE function trajectory_add(trajectory, "position") RETURNS trajectory
AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so', 'trajectory_add'
LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR +(
    leftarg = trajectory,
    rightarg = "position",
    procedure = trajectory_add,
    commutator = +
);

CREATE function segment(trajectory, int, int) RETURNS trajectory AS
'/usr/include/postgresql/9.1/server/lib/trajectory.so',
'trajectory_segment'
LANGUAGE C IMMUTABLE STRICT;

CREATE function segment(trajectory, timestamp, timestamp) RETURNS
trajectory AS

```

```

'/usr/include/postgresql/9.1/server/lib/trajectory.so',
'trajectory_segment_interval'
LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION length(trajectory trajectory) RETURNS float AS $$
DECLARE
    result float;
BEGIN
    --cast to geography for use projected into EPSG:4326
    (longitude/latitude)
    result := St_Length(geography(geometry(trajectory)));
    RETURN result;
END;
$$ LANGUAGE 'plpgsql';

CREATE FUNCTION avg_length(trajectory trajectory) RETURNS float AS $$
DECLARE
    result float;
BEGIN
    result := length(trajectory) / npositions(trajectory);
    RETURN result;
END;
$$ LANGUAGE 'plpgsql';

CREATE FUNCTION top_length(trajectory trajectory) RETURNS float AS $$
DECLARE
    i integer;
    length float;
    nposs integer;
    top_length float;
BEGIN
    i := 1;
    top_length := 0;
    nposs := npositions(trajectory);
    IF nposs > 1 THEN
        WHILE i < nposs
        LOOP
            length := length(segment(trajectory, i-1, i));

            IF length > top_length THEN
                top_length := length;
            END IF;
            i := i + 1;
        END LOOP;
    END IF;
    RETURN top_length;
END;
$$ LANGUAGE 'plpgsql';

CREATE FUNCTION low_length(trajectory trajectory) RETURNS float AS $$
DECLARE
    i integer;
    length float;
    nposs integer;
    low_length float;
BEGIN
    i := 1;
    low_length := -1;
    nposs := npositions(trajectory);
    IF nposs > 1 THEN

```

```

        WHILE i < nposs
        LOOP
            length := length(segment(trajjectory, i-1, i));

            IF length < low_length or low_length = -1 THEN
                low_length := length;
            END IF;
            i := i + 1;
        END LOOP;
    END IF;
    IF low_length = -1 THEN
        low_length := 0;
    END IF;
    RETURN low_length;
END;
$$ LANGUAGE 'plpgsql';

CREATE FUNCTION duration(trajjectory trajjectory) RETURNS float AS $$
DECLARE
    result float;
BEGIN
    -- return in seconds
    result := EXTRACT(EPOCH FROM (get_timestamp(trajjectory_end(trajjectory))
- get_timestamp(trajjectory_start(trajjectory))));
    RETURN result;
END;
$$ LANGUAGE 'plpgsql';

CREATE FUNCTION top_duration(trajjectory trajjectory) RETURNS float AS $$
DECLARE
    i integer;
    duration float;
    nposs integer;
    top_duration float;
BEGIN
    i := 1;
    top_duration := 0;
    nposs := npositions(trajjectory);
    IF nposs > 1 THEN
        WHILE i < nposs
        LOOP
            duration := duration(segment(trajjectory, i-1, i));

            IF duration > top_duration THEN
                top_duration := duration;
            END IF;
            i := i + 1;
        END LOOP;
    END IF;
    RETURN top_duration;
END;
$$ LANGUAGE 'plpgsql';

CREATE FUNCTION low_duration(trajjectory trajjectory) RETURNS float AS $$
DECLARE
    i integer;
    duration float;
    nposs integer;
    low_duration float;
BEGIN
    i := 1;

```

```

low_duration := -1;
nposss := npositions(trajjectory);
IF nposss > 1 THEN
    WHILE i < nposss
    LOOP
        duration := duration(segment(trajjectory, i-1, i));

        IF speed < low_duration or low_duration = -1 THEN
            low_duration := duration;
        END IF;
        i := i + 1;
    END LOOP;
END IF;
IF low_duration = -1 THEN
    low_duration := 0;
END IF;
RETURN low_duration;
END;
$$ LANGUAGE 'plpgsql';

CREATE FUNCTION avg_duration(trajjectory trajjectory) RETURNS float AS $$
DECLARE
    result float;
BEGIN
    -- return in seconds
    result := duration(trajjectory) / npositions(trajjectory);
    RETURN result;
END;
$$ LANGUAGE 'plpgsql';

CREATE FUNCTION avg_speed(trajjectory trajjectory) RETURNS float AS $$
DECLARE
    result float;
    duration float;
    length float;
BEGIN
    result := 0;
    duration = duration(trajjectory);
    length = length(trajjectory);

    IF duration > 0 THEN
        --meters/second to kilometers/hour
        result := (length/duration) * 3.6;
    END IF;

    RETURN result;
END;
$$ LANGUAGE 'plpgsql';

CREATE FUNCTION top_speed(trajjectory trajjectory) RETURNS float AS $$
DECLARE
    i integer;
    speed float;
    nposss integer;
    top_speed float;
BEGIN
    i := 1;
    top_speed := 0;
    nposss := npositions(trajjectory);
    IF nposss > 1 THEN
        WHILE i < nposss

```

```

        LOOP
            speed := avg_speed(segment(trajecory, i-1, i));

            IF speed > top_speed THEN
                top_speed := speed;
            END IF;
            i := i + 1;
        END LOOP;
    END IF;
    RETURN top_speed;
END;
$$ LANGUAGE 'plpgsql';

CREATE FUNCTION low_speed(trajecory trajecory) RETURNS float AS $$
DECLARE
    i integer;
    speed float;
    nposs integer;
    low_speed float;
BEGIN
    i := 1;
    low_speed := -1;
    nposs := npositions(trajecory);
    IF nposs > 1 THEN
        WHILE i < nposs
        LOOP
            speed := avg_speed(segment(trajecory, i-1, i));

            IF speed < low_speed or low_speed = -1 THEN
                low_speed := speed;
            END IF;
            i := i + 1;
        END LOOP;
    END IF;
    IF low_speed = -1 THEN
        low_speed := 0;
    END IF;
    RETURN low_speed;
END;
$$ LANGUAGE 'plpgsql';

CREATE FUNCTION positions(trajecory trajecory) RETURNS SETOF "position"
AS $$
DECLARE
    i integer;
    nposs integer;
BEGIN
    i := 0;
    nposs := npositions(trajecory);
    WHILE i < nposs
    LOOP
        RETURN NEXT (position_n(trajecory, i));
        i := i + 1;
    END LOOP;
    RETURN;
END;
$$ LANGUAGE 'plpgsql';

CREATE FUNCTION cb_smot(trajecory trajecory, threshold float, minduration
int) RETURNS SETOF trajecory AS $$
DECLARE

```

```

i integer;
stop integer;
speed float;
npos integer;
BEGIN
i := 1;
stop := -1;
npos := npositions(trajjectory);
IF npos > 1 THEN
    WHILE i < npos
    LOOP
        IF stop = -1 THEN
            speed := avg_speed(segment(trajjectory, i-1, i));
            IF speed <= threshold THEN
                stop := i-1;
            END IF;
        ELSE
            speed := avg_speed(segment(trajjectory, stop, i));
            IF speed > threshold THEN
                IF stop < i - 1 AND duration(segment(trajjectory, stop,
i-1)) > minduration THEN
                    RETURN NEXT (segment(trajjectory, stop, i-1));
                END IF;
                stop := -1;
            END IF;
        END IF;
        i := i + 1;
    END LOOP;

    IF stop <> -1 AND stop < npos-1 AND duration(segment(trajjectory,
stop, i-1)) > minduration THEN
        RETURN NEXT (segment(trajjectory, stop, npos-1));
    END IF;
END IF;
RETURN;
END;
$$ LANGUAGE 'plpgsql';

```

Extensão do arquivo: **postgis.sql**

```

CREATE OR REPLACE FUNCTION geometry("position")
RETURNS geometry
AS '$libdir/postgis-2.1','position_to_geometry'
LANGUAGE 'c' IMMUTABLE STRICT;

CREATE OR REPLACE FUNCTION geometry(trajjectory)
RETURNS geometry
AS '$libdir/postgis-2.1','trajjectory_to_geometry'
LANGUAGE 'c' IMMUTABLE STRICT;

```