Free University of Bozen-Bolzano
Faculty of Computer Science

Fakultät für Informatik
Facoltà di Scienze e Tecnologie informatiche
Faculty of Computer Science

Master Thesis

A Relational Implementation of Multimodal Road Networks

by
Viktorija Šukvietytė

Supervisor: Johann Gamper
Co-supervisor: Theodoros Chondrogiannis

2016

Contents

Abstract

Nowadays, the number of systems based on maps that use geographical data to build applications for route searching or trip planning is greatly increased. However, it is not so easy to model data efficiently. The multimodal networks integrate networks of different type: pedestrian, bus, train network, etc., which come from different sources and have a different structure. Some of these networks may hold features such as timetables that must be considered when a trip is planned.

In this thesis, we propose a relational implementation of multimodal road networks. Our implementation is based on the time-dependent approach that models timetable information in public transport systems. The PostgreSQL database under Ubuntu Operating System (OS) was chosen as a platform for the implementation. We adopt the following popular standards of imported data: GTFS and VDV to import timetables of the public transport systems; OSM, PBF to import pedestrian networks.

We implement maintenance tools for this model to import or drop different modes. Also, we propose query services about the data model. In order to facilitate and simplify the usage of query services in various programming platforms, it was implemented as a PostgreSQL database set of functions written in SQL and PL/pgSQL languages.

We describe in detail our model for multimodal road networks stored in PostgreSQL. Also, a description of implemented tools is presented. In the future, this model could be applied in other relational Database Management Systems (DBMS).

Chapter 1

Introduction

1.1 Motivation

The current pace of life makes people rush and daily use various means of transport. The smart devices with mobile applications such as maps, GPS navigation, public transport timetables, become the most faithful guide of the life. Without the latest technologies it would be difficult to keep up with the rhythm of life. However, a rare user of these technologies raises a thought, in what way it has become possible to instantly find out the desired travel direction, the shortest path, the fastest trip or various transport timetables and routes; how and where this variety of information is stored, and how the flow of information is manageable.

The problem in finding the shortest path became relevant more than 50 years ago (it was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later). It has been shown that the task of finding the shortest path belongs to the NP-hard (unsolvable) class of problems, but this problem was studied and currently is still researched by many scientists. There are a number of methods (techniques) to address this challenge.

Searching for the best solution, we are faced with trade-offs. One of them is the choice between the compactness of data storage and high speed to access data. Obviously, if we save some space for storing data, we need more time to access the required information. The data operated quickly will consume a lot of space. In other words, we have to decide how to store the problem data: in the specific format, compressed OS files or in relational DBMS. Practically, this trade-off has a number of specific solutions of data storage for static road networks: in the specific format, compressed OS files and in relational DBMS. As an example of specific format files, OSM, PBF are the most popular formats for street networks, and PostGIS – to store spatial and geographical objects in the PostgresSQL database. A different situation arises solving more complex problems in time-depending networks. There are many solutions that store data in specific format OS files like GTFS or

VDV. Meanwhile, it is difficult to find similar solutions of storing data in relational DBMS. The challenge becomes even more complex for multimodal networks.

1.2 Contribution

In this paper, we provide our insights what has been done before for the optimal route planning as a solid starting point for our relational implementation of multimodal road networks. We review querying techniques, modelling approaches and common challenges.

A model of multimodal networks was implemented. To gain the fast and convenient information access, we have chosen an innovative way of data storage in the relational DBMS (PostgreSQL).

We developed a tool that allows user to select, import and load data of desirable pedestrian and public transport networks to the model and combine them. Comprehensive schema descriptions together with instances are provided.

Moreover, we implemented a database application program interface (DB API) intended for application developers to get information from this model in various environments easily and quickly. In the end, we have performed several synthetic tests to evaluate and prove that our model works effectively and correctly.

1.3 Structure of the Thesis

The thesis is organized as follows. In Chapter 2, we briefly review groups of querying techniques, modelling approaches of time-dependent networks and their challenges. The model realization, schemas, and data examples are described in Chapter 3. The developed tools are described in Chapters 4 and 5 of this paper. Chapter 6 discusses the evaluation of the work. Chapter 7 summarizes our insights and provides future perspectives. In the Appendix, we provide detailed a manual of query services.

Chapter 2

Background

In this chapter, we briefly observe what has been done currently for multimodal networks as the basis for a new data model.

Regardless of the choice of transport network, we always have to compute some form of shortest path queries. There are basic types of the shortest path problems [10]:

- In the point-to-point shortest path problem, a graph G, a source $s \in V$, and a target $t \in V$ are given as input, and must compute the length of the shortest path from s to t in G.
- The one-to-all problem is to compute the distance from the given vertex v to all the vertices in the graph.
- The many-to-many problem finds the shortest paths for all pairs of vertices.

However, finding the optimal route is an NP-hard problem. Therefore, there are not one but a lot of algorithm solutions that are constantly evolving. They include multicriteria and require compromises in query time, space usage, preprocessing efforts and other factors. Numerous shortest path algorithms for the road network are described by H. Bast et al. [2]. They are classified by techniques and can always be combined with additional speedup too.

2.1 Shortest Path Computing Techniques

Basic Techniques. The standard technique to find the shortest path between nodes in the graph is Dijkstra's algorithm [6]. It is a baseline for numerous techniques that preprocess the input graph to achieve speedups. The nodes can be, for example, different cities, crossroads, street intersections, etc. The edges weights are the distances between a pair of nodes connected by direct road. There are many modifications of this algorithm. The most

common variant takes the single node as a source and finds the shortest path to other nodes in the graph. It also can be used to find the shortest path between two given nodes. The algorithm maintains tentative distances that are initiated ∞ for unreached nodes and 0 for the source node s. It picks an unvisited node with the lowest distance, calculates the distance through it to each unvisited neighbour and updates the neighbour's distance if it is smaller. Then the process stops as soon as the shortest path to the target node t is reached or all distances for all the pairs of nodes are found.

In practice, one can reduce the search space and accelerate the process, using the bidirectional search. It runs forward and backward at the same time and stops when intersection contains the same vertex.

Goal-Directed Techniques. These methods allow preprocessing of network data in order to obtain information used to speedup queries. The aim is to direct the search towards the target t by pruning the edges that are not in the direction to t, and preferring edges that shorten the distance. The classic goal-directed algorithm is A* search [5]. It uses a lower bound function on vertices. This induces us to scan earlier vertices that are closer to the target t. Another method based on the Dijkstra algorithm is Arc Flags [9]. This approach is combined with partitioning. In the preprocessing stage, the algorithm divides the graph into balanced cells with a similar number of vertices and has a small number of boundary vertices. The boundary can be used to prune the search. If the search reaches the cell that contains the target t, it starts evaluating the arc flag. Otherwise, the search is pruned.

Separator-Based Techniques. Methods partition the graph into small subgraphs by a small separator [12]. One of techniques is based on the vertex separator, which is a subset of vertices. There is another technique that use thearc separator. These separators can be used to overlay graph G. The distance of arc added between the separator vertices is equal to the distance in G. Such new graphs are much smaller and can accelerate the queries. Separators can have hierarchies too. However, addition of more shortcut arcs may significantly increase the space usage and preprocessing time.

Hierarchical Techniques. These methods exploit hierarchical nature of the road network in one way or another. This is very useful if we have the source and target nodes far away from each other. In this case, it suffices to scan only highway nodes.

As an example of the hierarchy approach, Contraction Hierarchies (CH) are the technique that generates a multi-layered node hierarchy in the preprocessing stage [7]. The idea is to reduce the visited number of vertices by ranking them according to "importance". The method introduces new edges, so-called shortcuts, during the preprocessing butvisits only the arc leading to higher rank vertices.

Bounded-Hop Techniques. The main idea of techniques is to precompute pairwise distances of vertices by labelling vertices. In order to get the shortest path, we use already precomputed distances between the pairs of vertices instead of the input graph. A simple approach is to precompute distances between all the pairs of vertices. In this way, a single look-up of the table is sufficient.

The Labelling algorithm is one of the techniques that compute labels for each vertex of the graph [10]. For example, the label $L(u)$ consists of a set of vertices (hubs) from the vertex u together with their distances from u. Thus, the distance of any pair s and t can be determined by looking at their labels. The chosen labels also obey the cover property. It means that intersection of labels for any pair of vertices s and t must contain at least one vertex on the shortest path s–t. In the direct graphs, the labels associated with vertices are split into forward and backward labels. However, the space usage is higher than that of similar methods.

Another Transit Node Routing is one of the fastest speedup algorithms to compute the shortest path in the road network [1]. During preprocessing, it selects a set of transit nodes of arterial networks. The chosen set is crucial to the performance of the algorithm. Then the distances between transit nodes, and the distances to their access nodes as well as information that identifies whether the shortest path does not cross the transit nodes are stored in the tables.

2.2 Modelling of Time-Dependent Networks

It is desirable that applications for computing optimal paths in road networks use realistic scenarios like transport networks, which can be affected over time, incorporate traffic congestions, delays, closures, and many other factors. This makes the problem more complicated and time-dependent because the best route solution depends on departure time. However, the techniques considered above can be adapted and used for efficiency. That can be done by assigning the travel time function to (some of) edges, representing how long it takes to traverse them at each time of the day. Also, they only can be traversed at specific points in time.

In this scenario, the input is given as a timetable consisting of a set of bus stops or train platforms, a set of routes (bus or train lines) and a set of trips. The trips are given as pairs of (origin/destination) stops and (arrival/departure) times.

Dealing with this issue, it does not suffice to compute a single shortest path like in static networks. The first challenge is to build the graph $G = (V, A)$ from a timetable so that the shortest paths in G corresponded to the optimal journeys. There are two common approaches to do that.

The solutions of the time-expanded and time-dependent data model are presented and compared in [4, 11].

Time-Extended Model. The idea of the time-extended model is to build an event graph that "enrolls" time. The model creates a vertex for each event and it connects consecutive arrival and departure events by connecting arcs. All vertices at the same stop are connected in chronological order by transfer arcs to enable transfers between vehicles. The model has already modified versions that try to optimize the number of transfers, reduce redundant arcs, incorporate the minimum change time given by input, etc.

Time-Dependent Model. In contrast to the previous approach, the resulting time-dependent graph is less space-consuming. Instead of unrolling the timetable, the model encodes time dependencies by the travel function on the arcs. In the model vertices correspond to stops. For each stop p and route r that serves p there is a route vertex. Route vertices at p are connected to the common stop by arcs with a constant cost of minimum change time. The route arcs connect the subsequent route vertices and form their trips. Both models are represented in Fig. 2.1.

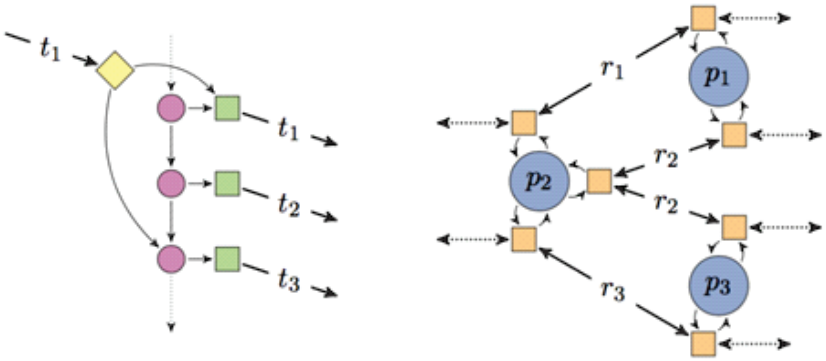

Figure 2.1. Time-expanded (left) and time-dependent (right) models. Trips t_i denote connection arcs in the time-expanded model, and routes r_i denote route arcs in the time-dependent model [2].

2.3 Querying Multimodal Time-Dependent Networks

When we are planning a journey from s to t, there are diverse criteria according to which we choose the "best" route. One of the goals is minimizing the travel time, i.e. to arrive as soon as possible to the destination point. Further, we face several problems during the modelling process:

Earliest Arrival Problem. The simplest problem is the earliest arrival problem [10]. A source stop, a target stop, and departure time are given. It is

necessary for a trip to start from the source stop no earlier than the given departure time and arrive at the target stop as soon as possible.

Range Problem. A similar variant is the range problem [2]. In this case, a time range replaces departure time and we need to set trips with the minimum travel time departing within the given range. Both this and the previous problems consider only arrival and departure times.

Multimodal Journey Planning. The multimodal networks are more complicated and require approximations [10, 3, 8, 14]. However, this is an increasingly relevant topic worldwide. They are composed of multiple modes: schedule-based transportations (bus, trains, flights) and unrestricted modes (walking, cycling, cars).

A common case to obtain a multimodal network is to build individual graphs for each transportation mode and merge into a single multimodal graph with link arcs. The shortest path computation in a multimodal network should incorporate historic patterns, rush hours, transit information, real-time traffic, user preferences and other constraints.

Transit Data Format. There are different size of public transport networks worldwide. In many cases, the agencies that manage these networks seldom communicate with one another. Even more, they do not have common agreements on the standards or formats for public transportation schedules. The situation changed in 2005, when the USA's public transit agency TriMet in collaboration with Google defined a common format for public transportation schedules and associated geographic information, so-called GTFS [13]. Since that time, the GTFS format has changed significantly and it is constantly evolving and having widespread effects among transit agencies. GTFS data are now being used by a variety of third-party software applications for many different purposes: trip planning, timetable creation, transit data visualization, to provide real-time transit information.

Chapter 3

Relational Implementation

In this chapter, we introduce a multimodal time-dependent theoretical network, which is a basis for our relational implementation. In the next sections, we show the model schemas in detail and examples of imported data.

3.1 Time-Dependent Data Model

In this section, we describe a theoretical time-dependent data model that represents the real processes of traffic in detail. We consider two basic types of modes: a pedestrian network and a bus network separately. These networks are connected through links. We define the structure of each network in the multimodal graph below.

Pedestrian Network. We define a pedestrian network as a direct graph $G_s = (V_s, E_s)$, where V_s is a set of vertices or nodes and E_s is a set of edges. Each node represents a real world object such as street intersects or crossroads. Edges represent a street or a part of it and store its source and destination nodes.

The connections among nodes are bidirectional, i.e. people can walk in both directions. Each node has a neighbourhood if there exists an edge going from this node to another node or there are any incoming edges to this node from other ones. The number of incoming edges is the so-called *in-degree* of sn_i node. The number of outgoing edges is an *out-degree* of pn_i node.

In Fig. 3.1, a small real world example is sketched. The solid lines are used to represent pedestrian paths. The nodes $pn_1, \ldots, pn_7$ represent street intersections, and are connected by pairs of street edges denoting the road segments.

Public Transport Network. This transport is a completely restricted mode. It has fixed routes, bounded by the timetable, and can be accessed from specific location (bus stop or station). In real life, a bus stops always at the pedestrian sidewalk so that passengers can wait there safely and reach it on

Figure 3.1. Pedestrian network.

foot. At the moment when a bus arrives, people can transfer from the pedestrian to transport network. Hence we can identify three main components as bus routes, bus stops, and other nongeometrical information needed as a bus schedule.

The structure of the public transport network is similar to the pedestrian network. It has an underlay graph $G_t = (V_t, E_t)$, where V_t and E_t are the set of nodes and edges, respectively. Each node denotes a station or bus stop in the real world for the route i and holds information about the routes and timetables.

As you can see in Fig. 3.2, a small real world example represents a transport mode. $bn_1, \ldots, bn_4$ are bus network nodes, and the dashed lines indicates different routes passing through the nodes.

Link Network. As a result, we construct a final multimodal graph by mapping the transport network on the pedestrian network. Bus stops or stations usually are at a distance from the street crossroads and intersection, i.e. the nodes of the pedestrian network. Therefore, we find the closest point on the edge to each given point (bus stop or station) as a fraction of total length, make a new node in that place, and link it to the closest stop. These links let easily transfer between both modes. In Fig. 3.3, a full multimodal network that combines the pedestrian and transport networks is shown. The blue dots are new street nodes as projections of bus stops $bn_1, \ldots, bn_4$.

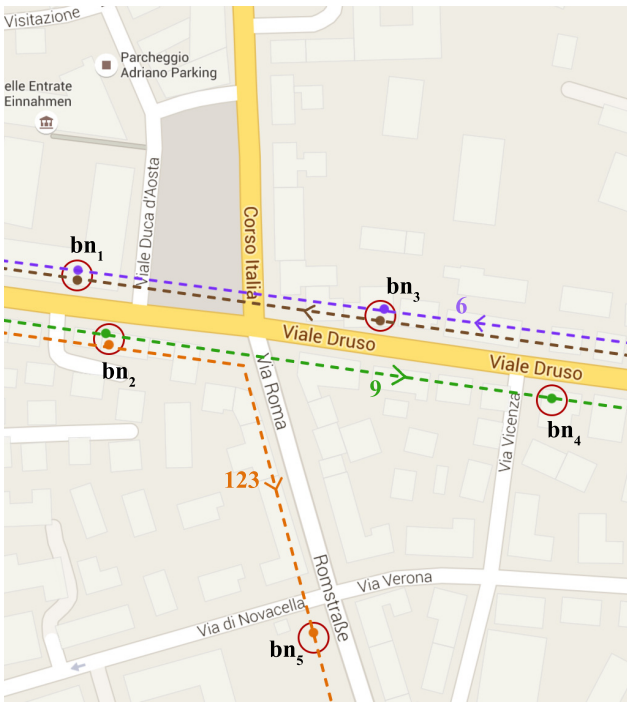

Figure 3.2. Transport network.

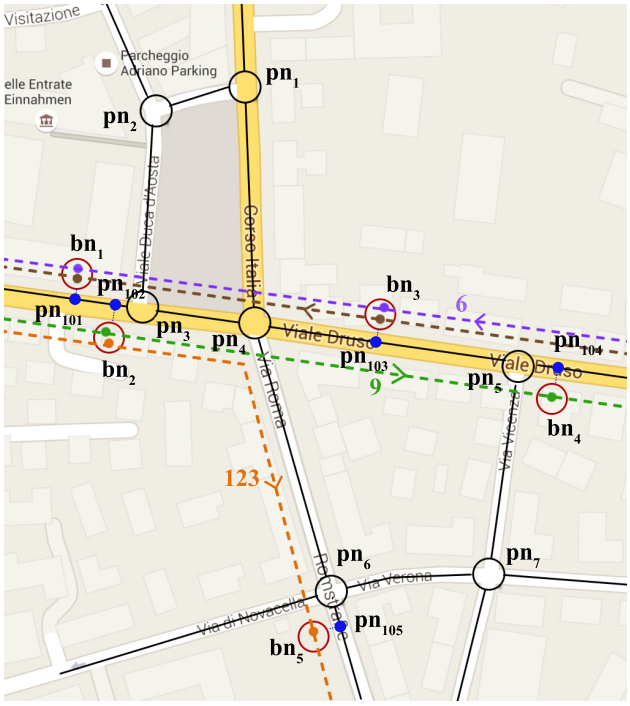

Figure 3.3. Multimodal network with link edges expressed in dotted lines.

3.2 Database Schema

In this section, we provide instances of our model tables and their brief descriptions.

3.2.1 Location Schema

The first table we imported to the data model stores information about all pedestrian networks. It is primarily identified by the column *loc_id*. The column *loc_short_name* is a name of region and *loc_long_name* shows a country and continent, where region is located. *Loc_url* is a download link of appropriate OSM.PBF file. We also use two more columns *url_file_name* and *url_file_type* to have a file name and its type separately. In addition, table stores geometry of location, download, parse and import statuses. Schema of the table *Location* with all columns and types is represented in Fig. 3.4.

location	
loc_id	serial[10]
loc_short_name	varchar[2147483647]
loc_long_name	varchar[2147483647]
loc_url	varchar[2147483647]
url_file_name	varchar[2147483647]
url_file_type	varchar[2147483647]
loc_geometry	geometry[2147483647]
loc_geometry_str	varchar[2147483647]
loc_poly_file	varchar[2147483647]
download_directory	varchar[2147483647]
download_status	bool[1]
parse_directory	varchar[2147483647]
parse_status	bool[1]
import_status	bool[1]
< 0 4 rows	1 >

Figure 3.4. Schema of the table *Location*.

The schema of locations is defined as follows:

Location(loc_id, loc_short_name, loc_long_name, loc_url, loc_file_name, loc_file_type, loc_geometry, loc_geometry str, loc_poly_file, download_directory, download_status, parse_directory, parse_status, import_status)

3.2.2 Network Schema

The structure of the table *Network* is similar to that of the table *Location*. Each network is identified by the column *net_id*, which is a primary key. It contains columns for names, download link, type, geometry, and the rest

information as in the table *Location*. In addition, this table has the column *link_status* of Boolean type. It shows whether the network is linked with any region. The schema of table *Network* with all the columns and types is represented in Fig. 3.5.

network	
net_id	serial[10]
net_short_name	varchar[2147483647]
net_long_name	varchar[2147483647]
net_url	varchar[2147483647]
url_file_name	varchar[2147483647]
url_file_type	varchar[2147483647]
net_geometry	geometry[2147483647]
download_directory	varchar[2147483647]
download_status	bool[1]
parse_directory	varchar[2147483647]
parse_status	bool[1]
import_status	bool[1]
link_status	bool[1]
< 0 3 rows	5 >

Figure 3.5. Schema of the table *Network*.

The schema of networks is defined as follows:

Network(net_id, net_short_name, net_long_name, net_url, net_file_name, net_file_type, net_geometry, download_directory, download_status, parse_directory, parse_status, import_status, link_status)

3.2.3 Pedestrian Network Schema

The pedestrian network described in Section 2.2, is imported into a database. They are intended to store the nodes and edges of the pedestrian network.

Street_node. Each node is the start or end point of the edge and is identified by a primary key constraint on the *node_id* that is of serial type. The column *osm_id* originally came from the OpenStreetMap (OSM) file that has a public access and further could be used for updates and extensions of the map. The column *node_location* contains spatial information of the node, i.e. longitude and latitude coordinations. *loc_id* denotes the location that holds the node.

The schema of the street nodes is defined as follows:

Street_node(node_id, osm_id, node_location, loc_id)

Street_edge. A table presents a list of streets. Each row is identified by a primary key constraint on the column *edge_id* that is of serial type. There are *edge_source* and *edge_target* references to the table *Street_node* that show

the nodes as start and end points of the edges. The column *edge_length_km* shows the real length of a street in kilometres. The column *edge_geometry* is a street topology, i.e. spatial information of an edge as linestring. *loc_id* indicates a location that holds an edge.

The schema of the street edges is defined as follows:

Street_edge(edge_id, edge_source, edge_target, edge_length_km, edge_geometry, loc_id)

In Fig. 3.6, we represent *Street_node* and *Street_edge* relations by an ER-diagram, including all the columns, types, primary and foreign keys as well as the number of rows in each table.

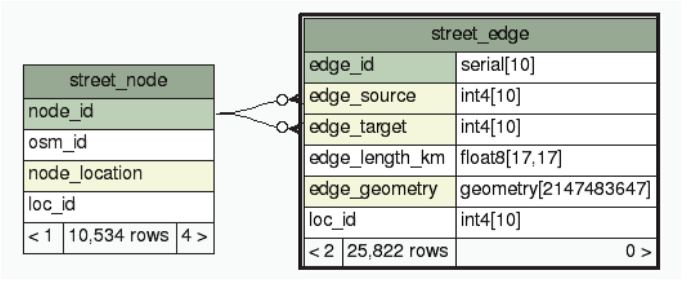

Figure 3.6. Pedestrian network schema.

3.2.4 Transport Network Schema

The transport network described in Section 2.2, is similarly imported into a database as the pedestrian network. The network includes relations of bus nodes and edges, relations of schedules and one relation of geometric segments of routes. See the relations of transport network nodes and edges in Fig. 3.7.

Busnet_node. Relation stores all information about the bus node as a station. Each node is identified by the primary key on the column *node_id* that is of serial type. Other columns: *stop_id, route_id* and *street_node_id* are, respectively, integer foreign keys to these tables: *Bus_stop, Bus_route, Street_node*.

The schema of the bus network nodes is defined as follows:

Busnet_node(node_id, stop_id, route_id, street_node_Id)

Busnet_edge. The table of edges represents movements among the bus nodes. Each row of this table is primarily identified by the column *edge_id* that is also of serial type. Both integer *edge_source* and columns *edge_target* are foreign keys to the table *Busnet_node*. They show the start and end of the edge. The column *route_id* is the integer foreign key to the table *Bus_route*. The column *service_id* checks the validity of edge in certain day. It is also the integer foreign key and reference to the table *Bus_service*. The last column

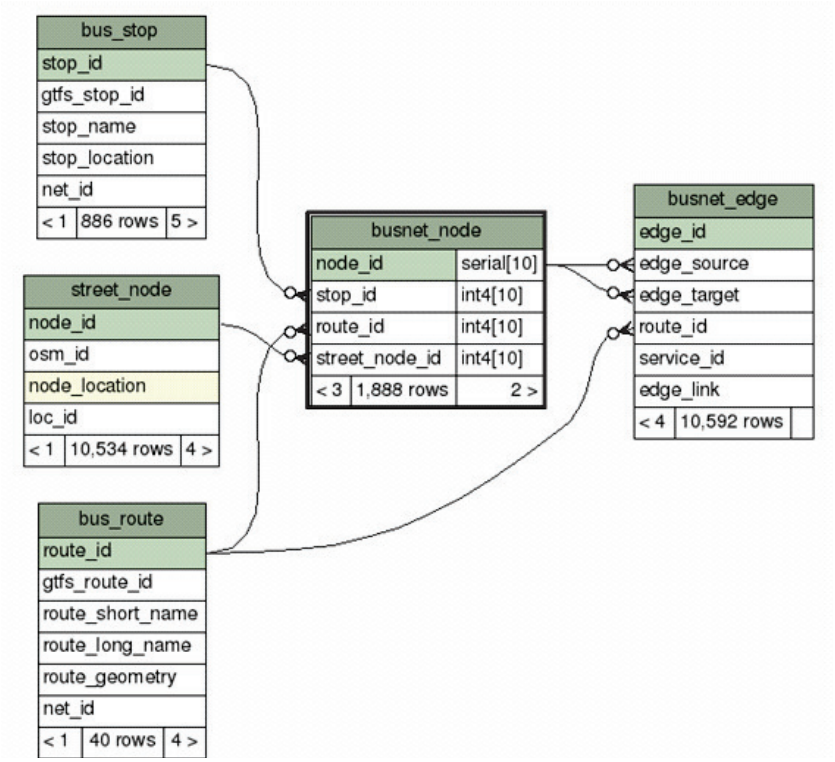

Figure 3.7. Transport network schema.

edge_link is of Boolean type and shows whether the edge passes through the same station or different ones.

The schema of the bus network edges is defined as follows:

Busnet_edge(edge_id, edge_source, edge_target, route_id, service_id, edge_link)

The tables *Bus_route, Bus_trip, Bus_stop, Bus_stop_times, Bus_service, Bus_service_date* intended to store the imported GTFS data, are presented below. Each table has a one-to-one correspondence with GTFS. Also, tables have an additional field to store the original GTFS IDs with exception of the table *Bus_stop_times*. The extra attribute together with the column *net_id* explicitly show the relation between GTFS data and the information contained into our model.

Bus_route. The table has information about bus routes. Each route is identified by the primary key on the column *route_id* that is of integer type. The column *gtfs_route_id* is of integer type and comes as the original ID from the source file. The route has the string type columns *route_short_name* and *route_long_name* of route short and long names. It also has geometry and network ID, to which the bus route belongs.

The schema of the routes is defined as follows:

Bus_route(route_id, gtfs_route_id, route_short_name, route_long_name, route_geometry, net_id)

Bus_trip. The primary key on the column *trip_id* that is of integer type identifies each trip of the route. In addition, the table consists of two more integer columns *route_id* and *service_id*. The latter checks the validity of a trip in particular days. Both columns are foreign keys. The column *gtfs_trip_id* is of integer type and comes as the original ID from the source file. The column *net_id* shows a network ID, to which the bus trip belongs.

The schema of the trips is defined as follows:

Bus_trip(trip_id, route_id, gtfs_trip_id, service_id, net_id)

Bus_stop_times. Each row is identified by the primary key compounded of three columns: *trip_id, stop_id* and *stop_sequence*. All of them are of integer type and at the same time *trip_id* and *stop_id* are foreign keys, respectively, to tables *Bus_trip* and *Bus_stop*. The column *stop_sequence* indicates the sequence number of stops in the trip. The arrival and departure times are given in seconds and stored in integer columns *arrival_time* and *departure_time*.

The schema of the stop times is defined as follows:

Bus_stop_times(trip_id, stop_id, arrival_time, departure_time, stop_sequence, net_id)

Bus_stop. The table contains information about the bus stops. Each of them is identified by the primary key on the column *stop_id* that is of integer type. The column *gtfs_stop_id* is of integer type and comes as the original ID from the source file. The bus stop has a name in the string column *stop_name* and spatial information, i.e. 2D coordinates of points in the map, in the geometric column *stop_location*. The column *net_id* shows network ID to which the stop belongs.

The schema of the stops is defined as follows:

Bus_stop(stop_id, gtfs_stop_id, stop_name, stop_location, net_id)

Bus_service. A relation is meant to check the validity of trips according to week days or exceptional dates. Each row identified by the primary key on the column *service_id*. *gtfs_service_id* is of integer type and comes as the original ID from the source file. The rest columns are of Boolean type indicating a certain week day. The columns *start_date* and *end_date* are of date type to indicate a certain date. The column *net_id* shows the network to which each service belongs.

The schemas of the services are defined as follows:

Bus_service(service_id, gtfs_service_id, net_id)

Bus_service_date(service_id, monday, tuesday, wednesday, thursday, friday, saturday, sunday, start_date, end_date, exception_type)

A relational schema of GTFS data is depicted in Fig. 3.8.

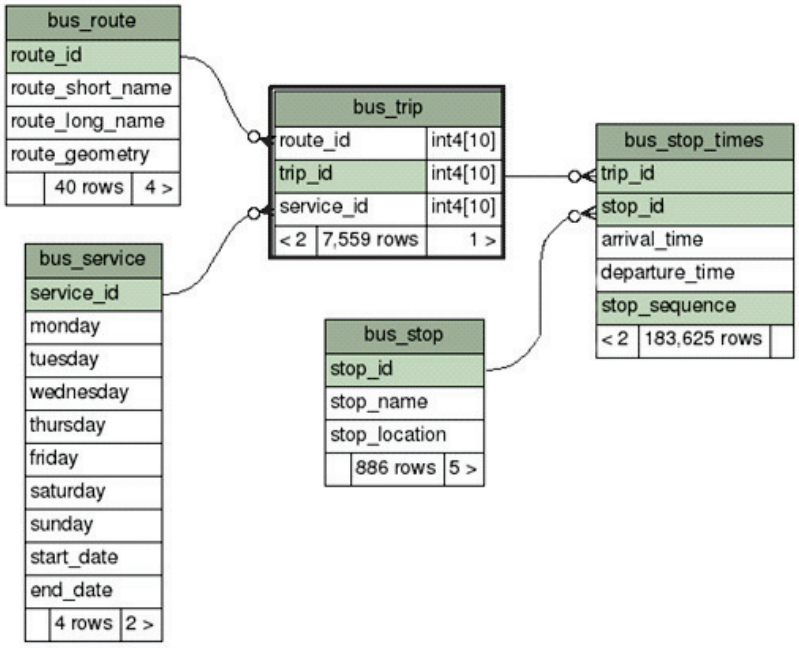

Figure 3.8. Relational schema of GTFS tables.

Bus_route_segment_geom. The table is meant for visualization purposes. The primary key on the column *segment_id* identifies a segment of the bus route that is of serial type. Integer columns *top_source* and *stop_target* are the bus stop IDs. They refer to the table *Bus_stop*, respectively. The column *route_id* is of integer type and refer to the table *Bus_route*. *Segment_geometry* is a geometric column of a linestring shape. A relational schema of transport route segment geometries is depicted in Fig. 3.9.

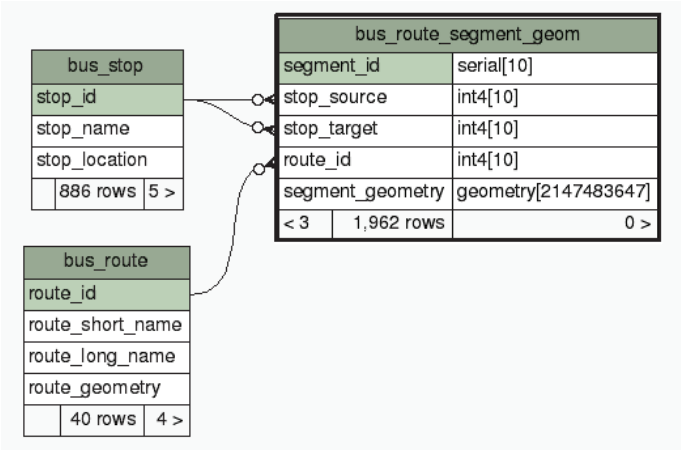

Figure 3.9. Relational schema for transport route segment geometries.

The schema of the route segments geometries is defined as follows:

Bus_route_segment_geom(segment_id, stop_source, stop_target, route_id, segment_geometry)

3.2.5 Link Network Schema

This section reviews a schema of the link network described in Section 2.2. The schema consists of only one table that combines pairs of nodes between *Street_node* and *Bus_stop* relations.

Bus_street_link. The relation is primarily identified by the column *link_id* that is of serial type. Each row combines bus stops with street nodes, and stores the references to the corresponding tables *Bus_stop* and *Street_node*.

The schema of the links is defined as follows:

Bus_street_link(link_id, bus_stop_id, street_node_id, link_geometry)

A relational schema of links is depicted in Fig. 3.10.

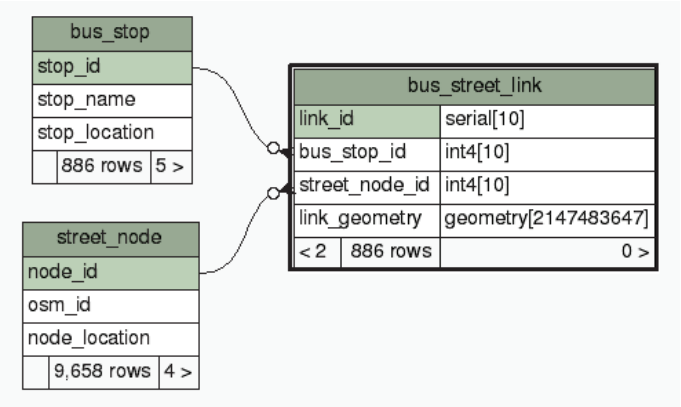

Figure 3.10. Relational schema of links.

3.3 Examples

In this section, we illustrate some examples of real data of the tables stored in our data model as well as description of the information showed.

Locations. Table 3.1 contains the data about regions that we can import to our model. Each area has a geometry, which is a specified value from Extended Well-Known Text representation (EWKT). EWKT is one of PostGIS formats to represent the geometric objects with alphanumeric values. *Download_status*, *parse_status* and *import_status* are Boolean attributes and indicate statuses of locations.

Table 3.1. Instance of the table *Location*.

Entity	Record 1	Record 2
loc_id	1	2
short_name	Barcelona	Bilbao
long_name	Europe/Spain	Europe/Spain
loc_url	http://download.geofabrik...	http://download.geofabrik...
url_file_name	spain.osm.pbf	spain.osm.pbf
url_file_type	OSM.PBF	OSM.PBF
loc_geometry	0103000000010...	0103000000010...
loc_geometry_str	POLYGON((−71.1776585...	POLYGON((−71.1776585...
osm_file_name	barselona.osm	bilbao.osm
poly_file_name	barselona.poly	bilbao.poly
pf_context	Barcelona...	Bilbao...
download_dir	./data	./data
d_status	FALSE	FALSE
...	...	...
import_status	FALSE	FALSE

Networks. In Table 3.2, the data about schedules are shown similarly as in the table *Location*. Each network contains geometry represented in EWKT format. This shape is constructed during the import phase and covers all bus stops of the specific network. Among the Boolean attributes, this table has *link_status* that indicates whether the networks are already linked with regions.

Table 3.2. Instance of the table *Network*.

Entity	Record 1	Record 2
net_id	1	2
short_name	MeranoBolzano	Trenitalia
long_name	Europe/Italy	Europe/Italy
net_url	http://open.sasabz.it/files...	http://www.gtfs-data-exch...
url_file_name	meran_vdv.zip	trenitalia_gtfs.zip
url_file_type	VDV.ZIP	GTFS.ZIP
net_geometry	010300002073...	
download_dir	./data	./data
d_status	FALSE	FALSE
...	...	...
import_status	FALSE	FALSE
link_status	FALSE	FALSE

Pedestrian Nodes. The street points are stored in a table *street_nodes*, as you can see in Table 3.3. In Section 4.3, we have mentioned that new unique IDs are assigned to nodes. We choose the bigint type for the original IDs that can be even larger than the attribute *node_id*. The points have coordinates and they are stored in EWKT format as well.

Table 3.3. Instance of the table *Street_node*.

node_id	osm_id	node_location	loc_int
1	9192415	0101000020...	4
2	9194517	0101000020...	4
3	9195242	0101000020...	4
...	...	...	...

Pedestrian Edges. Once we have the points of location, we use them to build edges. Some rows of the table *street_edges* are shown in Table 3.4. Under the field *edge_lengh_km* there are double precision values with 8 decimal digits. This information is obtained from OSM source files. The edge geometry is stored in EWKT format.

Table 3.4. Instance of the table *Street_edge*.

edge_id	edge_source	edge_target	edge_length_km
1	1	7	0.20901966
2	2	8546	0.10151595
3	3	4	0.13601428
...	...	...	...
edge_id	edge_geometry	loc_id	
1	0102000020E61...	4	
2	0102000020E61...	4	
3	0102000020E61...	4	
...	...	...	

Bus Network Nodes. Timetable nodes are saved in the database similar to the pedestrian ones. Each node stores three integer type attributes: *stop_id*, *route_id* and *street_node_id*. An instance of this table is shown in Table 3.5.

Table 3.5. Instance of the table *Busnet_node*.

node_id	stop_id	route_id	street_node_id
...	...	...	...
30	5	17	10521
31	6	17	10570
32	7	17	10392
...	...	...	...

Bus Network Edges. In Table 3.6, we have an instance of the table *Bus_net_edge*. There are no attributes to geometry or length. Each row stores start and end nodes of edges. We have the attribute *route_id* that denotes the edge belonging to the route. The attribute *service_id* shows the validity of edge in specific days. The link attribute indicates type of links. If the *edge_link* value is FALSE, the edges are between different bus stops and their

source and target node have the same route. If the *edge_link* value is TRUE, the source and target nodes are connected by edges in the same stop and have different routes.

Table 3.6. Instance of the table *Busnet_edge*.

edge_id	edge_source	edge_target	route_id	service_id	link
1	607	601	20	4	FALSE
2	1329	1376	38	3	FALSE
3	1233	1237	38	4	FALSE
...	...	...	...	...	...

Bus Routes. An instance of *Bus_routes* is shown in Table 3.7. This is a small table, where *route_id* and original GTFS IDs are the most important information in our model. *Route_short_name* and *Route_long_name* are useful for query functions and user interface.

Table 3.7. Instance of the table *Bus_route*.

route_id	gtfs_route_id	r_short_name	r_long_name	...	net_id
...	...	...	...	...	...
5	6	6 ME	6	...	1
6	110	110 BZ	110	...	1
7	111	111 BZ	111	...	1
...	...	...	...	...	...

Bus Stops. In Table 3.8, some rows of the table *Bus_stop* are shown. This table has original IDs referred to OSM sources. There is a attribute *stop_location* to coordinates of bus stops. Also, the names of stops are stored in this relation.

Table 3.8. Instance of the table *Bus_stop*.

stop_id	gtfs_stop_id	stop_name	stop_location	net_id
...	...	...	...	...
10	248	Via delle Corse - Rennweg	0101000020E61...	1
11	249	Ospedale - Krankenhaus	0101000020E61...	1
12	306	Terme - Thermen	0101000020E61...	1
...	...	...	...	...

Bus Services. Our model has 2 tables assigned to services. The data example of the table *Bus_service* is shown in Table 3.9. We have Boolean attributes to indicate valid days for routes and the period displayed in the attributes *start_date* and *end_date*. The table *Bus_service* contains IDs of services: globals and original GTFS IDs, and *net_ID* values.

Table 3.9. Instance of the table *Bus_service*.

service_id	gtfs_service_id	net_id
1	21	1
2	23	1
3	24	1
...	...	...

Table 3.10. Instance of the table *Bus_service_date*.

service_id	Monday	...	Sunday	start_date	end_date	exception_type
1	FALSE	...	TRUE	2015-06-21	2015-08-30	1
2	TRUE	...	FALSE	2015-06-22	2015-09-03	1
3	FALSE	...	FALSE	2015-06-27	2015-09-05	2
...	...	...	...	...	...	...

Bus Stop Times. Table 3.11 describes all the stops of trips and their arrival and departure times. These instances are defined as the number of seconds in an internal format. The field *stop_sequence* indicates the sequence number of the bus stop in the trip.

Table 3.11. Instance of the table *Bus_stop_times*.

stop_id	trip_id	arrival_time	departure_time	stop_sequence	net_id
...	...	...	...	...	...
885	2852	30060	30060	2	1
885	8995	25200	25200	21	1
885	8994	35160	35160	21	1
...	...	...	...	...	...

Bus Trips. An instance of the table *Bus_trip* is depicted in Table 3.12. Here we have a trip ID of integer type together with a route ID and service ID. The attribute *gtfs_trip_id* is original IDs from OSM source files.

Table 3.12. Instance of the table *Bus_trip*.

trip_id	route_id	gtfs_trip_id	service_id	net_id
1	1	3459	2	1
2	1	3458	2	1
3	1	3457	2	1
...	...	...	...	...

Bus Route Segment Geometries. This table is meant for visualization purposes. The data example shown in Table 3.13 represents all the segment geometries of routes in EWKT format. Each segment has source and target nodes of an edge as well as ID of the route that passes this segment.

Table 3.13. Instance of the table *Bus_route_segment_geom*.

segment_id	stop_source	stop_target	route_id	segment_geom
1	1	11	24	0102000020...
2	1	12	13	0102000020...
3	1	149	1	0102000020...
...	...	...	...	...

Bus-Pedestrian Links. The last table of the schema is the most important in order to construct the final multimodal network. In each entry, *bus_stop_id* of the table *Bus_net_node* is associated with that of the table *Street_node* to build a specific link edge. These links also have geometry. One can see some rows in Table 3.14.

Table 3.14. Instance of the table *Bus_street_link*.

link_id	bus_stop_id	street_node_id	link_geometry
1	718	9813	0102000020...
2	754	10620	0102000020...
3	516	10562	0102000020...
...	...	...	...

Chapter 4

Import of Multimodal Network

The development process of the multimodal network can be split into several stages. These stages are independent processes that can be executed separately or in combination with other model building processes. The model building principles are essential to understand in order to obtain the final fully functioning multimodal network. We split the model building process into these stages:

1. Creation and initialization of tables in database;
2. Import of the pedestrian network;
3. Import of the transport network;
4. Linking of pedestrian and transport networks;
5. Implementation of query services (DB API).

First of all, we briefly explain the purposes of each stage. Later on, we describe each stage in detail.

4.1 Creation and Initialization of Tables in Database

This stage is dedicated to create or recreate multimodel network tables and indices in the database. As a result, we obtain the new empty model tables and indices. The rest old data are deleted. This stage must be performed prior to all the other stages, except for the fifth stage. The creation of DB API can be performed at any time.

In addition to tables and indices, the data for tables *Location* and *Network* are also loaded in this stage. The latter tables store information about pedestrian and transport networks, such as download URL, original file name in Internet, file type, download category, download, parse, import statuses, etc. The data are predefined in the planning process about the imput of certain networks to the multimodal model.

4.2 Import of the Pedestrian Network

This stage is dedicated to import selected predefined pedestrian networks to a multimodal network. It includes download of OpenStreetMap (OSM) free data from Internet [15]. OSM is a collaboration project that contributes and maintains roads, trails and much more, all over the world. It supports some different formats like .pbf, .bz2 and shape file. For convenience, in our implementation we used only the PBF format.

The creation of a .poly file is the next thing included in the import phase. The required region is extracted from the whole country. Then we parse and load data into a database. This process can be carried out as long as we want and stop the execution at any point of the phase. Successfully completed steps are remembered and there is no need to repeat them any more. In the same time, we import only one pedestrian network. The previous stage, i.e. the model tables, is prerequisite in order to run the current one.

A pseudocode is the best way to describe this process:

Algorithm 1. Algorithm for the pedestrian network import

Require: Select Pedestrian Network net_id

1: Read information from the table *Location* about net_id
2: Test whether the file is downloaded
3: **if not** a downloaded file exists **then**
4: Download the file from Internet with the command wget
5: **end if**
6: Test whether there exists a .poly file
7: **if not** .poly file exists **then**
8: Create a .poly file
9: **end if**
10: Osmosis tools extract a region defined by a .poly file from the OSM.PBF file
11: Osm2po converter parses the OSM file we have got after step 10
12: The result obtained at step 11 is inserted into a temporary table *osm_2po_4pgr*
13: Data are loaded from the temporary table to the corresponding multimodal network tables

It should be noted that we import all pedestrian networks to the same tables. The street nodes and edges assigned to unique IDs during the import process and the original ones are stored as extra attributes. Moreover, additional information is stored about the network.

In this stage, we use three external tools: GNU wget, Osmosis tools, and Osm2po converter [16]. GNU wget is a free softer package for retrieving files using HTTP, HTTPS and FTP, and the most widely used Internet protocols.

Osmosis is a command line Java application for the processing OSM data. Osm2po is a converter and routing engine for OpenStreetMap.

4.3 Import of the Transport Network

This stage is dedicated to import the predefined selected public transport networks to the multimodal network. It includes download of public transportation schedules and associated geographic information in VDV [19] or General Transite Feed Specification (GTFS) [18] formats as well as the conversion of VDV to GTFS format and loading data to database. Simultaneously, only one transport network is imported to multimodal network. The first stage is prerequisite to run it.

Similarly as the pedestrian network, all transport networks are imported to the same tables, but different than for the pedestrian network. The edges and nodes are assigned to unique IDs during the import process and their original IDs remains as extra attributes. Moreover, additional information is stored about the network.

Algorithm 2. Algorithm for the transport network import

Require: Select Transport Network net_id
1: Read information from Network table about net_id
2: Test whether the file is downloaded
3: **if not** a downloaded file exists **then**
4: Download file from Internet with command wget
5: **end if**
6: Check whether the downloaded file is in VDV or GTFS format
7: **if** the format of file is VDV **then**
8: Convert VDV452 to GTFS format, using OneBusAway
9: **end if**
10: Parse GTFS files and load to relevant multimodal network tables for transport network
11: Load data into a database the rest tables: Bus_net_node, Bus_net_edge, $Bus\_net\_route\_segment\_geom$

In this stage, we use 2 external tools: GNU Wget and a converter from OneBusAway.

- GNU Wget that is a free software package for retrieving files using HTTP, HTTPS and FTP, the most widely-used Internet protocols.
- OneBusAway offers a suite of application programming interfaces (APIs) that facilitate and support the development of a wide range of third-party applications, based on the actual vehicle locations and on scheduled and predicted arrival times.

Line 8 in the pseudocode requires a further explanation. We use GTFS to define a common format for public transportation schedules and associates geographic information. GTFS was developed by Google to incorporate transit data into Google Maps. This standard is much simpler than VDV. A GTFS feed is a collection of at least 6 and up to 13 CSV files (with extension .txt):

agency: list of agencies, including names, websites, contact information, etc.

Required fields:

- agency_name
- agency_url
- agency_timezone

routes: identifies all distinct routes.

Required fields:

- route_id (primary key)
- route_short_name
- route_long_name
- route_type

trips: all the trips for the available route with valid days.

Required fields:

- trip_id (primary key)
- route_id (foreign key)
- service_id (foreign key)

stop_times: connections between two stops for a given trip with related arrival and departure times.

Required fields:

- stop_id (primary key)
- trip_id (foreign key)
- arrival_time
- departure_time
- stop_sequence

stops: defines the geographic locations for each bus stops or stations

Required fields:

- stop_id (primary key)
- stop_name

- stop_lon
- stop_lat

calendar and **calendar_dates:** defines service patterns that operate recurrently, for example, each weekend. A special event that does not repeat will be defined in the table *Calendar_dates*.

Required fields:

- service_id (primary key)
- monday
- tuesday
- wednesday
- thursday
- friday
- saturday
- sunday
- start_date
- end_date

4.4 Linking of Pedestrian and Transport Networks

In this stage, the pedestrian and transport networks are already imported and linked together. It is the last process enabling us to use the multimodal network. Simultaneously, only one pedestrian network and one transport network can be linked together. This fact requires the initial tables to be created in the database and at least one pedestrian and one transport network loaded in the database.

Algorithm 3. Algorithm for linking of pedestrian and transport networks

1: Set all possible pedestrian-transport linkings
Require: Select pedestrian-transport network $loc\_id\text{-}net\_id$
2: Street nodes obtained by projections of the closest stops are inserted.
3: Street edges are split through the new inserted nodes
4: Bus_net_nodes are appended with information obtained in step 2

Both the pedestrian and transport networks have the defined geographic area (coordinates). This information is stored in the model tables (*Location* and *Network*). At the first step of this stage, this information is used for intersection of networks. Thus, we get all possible links.

It is necessary to highlight that we had to find the closest streets (edges) to the bus stops (nodes) in the third step. Then, locations of the closest

points on edges to the nodes were found. The edges were split through these points into fractions.

Figure 4.1 depicts the situation before the splitting process. The picture below represents the situation after splitting. A, B, C, D correspond to street nodes. E and F correspond to bus stops. B and C are the closest points on edge AD to bus stops. Also, B and C are new street nodes that split edge AD into AB, BC, and CD fractions.

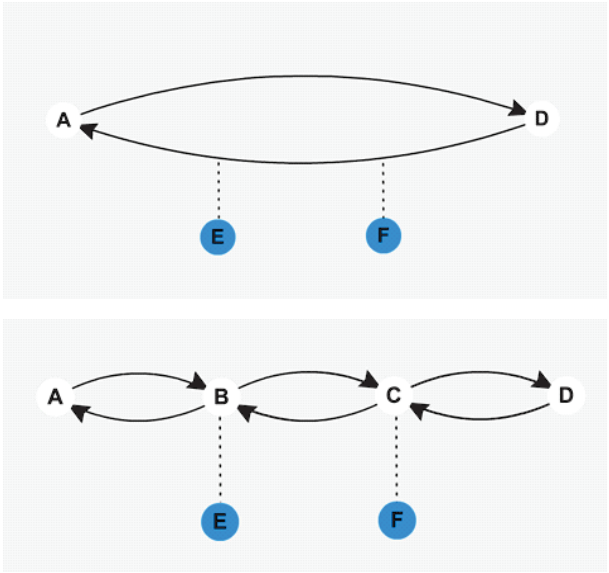

Figure 4.1. Splitting of street edges into fractions through projections of the closest stops.

4.5 Development Environment and Tools

The process to create our multimodal network has been finished and tested under Linux Ubuntu 14.04. A virtual machine was created based on Oracle VM VirtualBox in order to become more independent of the OS. Data of the multimodal network are stored in PostgreSQL 9.4 database [17]. Import tools use Java 9 SE, PostGIS 2.1, GNU Wget, Osmosis tools, osm2po converter, and a converter from the OneBusAway suite. The Entity-Relationship (ER) diagrams were generated using the SchemaSpy tool. The model was tested using the PostGIS extension, so-called pgRouting. We execute the individual script for each stage of the model building process. The scripts are written in SQL, PL/pgSQL languages. The whole flow of work is managed by the bash script *v_run-all.sh*. This script is carried out in the interactive mode.

Chapter 5

Query Services (DB API)

One of the objectives of the multimodal network building is understandable, easy, and fast access to information. Well knowing the internal structure of the model allows extracting the information about the model objects and their relationships to other model objects directly from the model tables. Such a way is quite complicated for an ordinary user or applications that use model data. In turn, this situation reduces a user base. In order to simplify the process and facilitate user's work to get data from the model, a set of functions was developed that releases him from writing complex queries. Functions can be called from very different environments like Bash, SQL, PERL, JAVA scripts, etc. Typically, these functions are called in the query form of:

SELECT * FROM DB_API_Function(Parameter_1, Parameter_2, ...)

Many functions can be called with different types and number of parameters. It facilitates customization of functions for different needs. A comprehensive description of functions with examples of their parameters are provided below.

As mentioned above, all functions can be classified into groups according to their purpose of use.

5.1 Implementation of Query Services (DB API)

The Database Application Interface (DB API) was implemented and loaded into a database. DB API consists of a set of functions to maintain the multimodal model. As stated above, this stage can be run at any time regardless of what stages have been done or not before. All DB API functions can be classified into groups according to their usage. These groups are shown in the following table.

Table 5.1. Function groups implemented to maintain the model.

Function name	Function description
Transport network static information	
MN_ST_list	List of stops by the network
MN_ST_of_R	Stops of route
MN_Route_names & MN_Net_routes	List of route names
MN_R_segments	List of route segments
Transport network information	
MN_Stop_seq_in_Trip	Stop sequence numbers in trip
MN_Stop_list_of_Trip	Stop list of a trip
MN_Stop_list_of_Route	Distinct stop lists of trips of a route
MN_R_ST_list	Name list of route stops of different trips
MN_R_ST_Time_list	Route departure times in a stop
Functions for the shortest path calculation	
MN_ST_NearestST_list	The list of the nearest stops from a given stop by the given distance
MN_R_Closest_stops	Full information on valid routes to the closest stops for a given date
MN_ClosestPoint	Projection of a bus stop on the closest street edge
MN_SplitEdge	Splits of the street edge through a given point

5.2 Static Information of the Transport Network

A group of functions that return static non time-dependent information about transport network objects are: bus stops, routes, etc. The following functions described below belong to this group.

MN_ST_list – the purpose of this function is to return the list of bus stops belonging to a particular network. The network ID has to be specified in the list of parameters.

MN_ST_of_R – this function returns ID numbers of all stops of the route. A route ID has to be specified in the list of parameters. A list of bus stops is not sorted, because the route can have several trips with the different list of stops.

MN_Route_names – this function returns names of the route (short, and long). The route ID has to be specified in the parameter list.

MN_Net_routes – this function returns all route names of a given network (short, long). The network ID has to be specified in the list of parameters.

MN_R_segments – this function returns a segment list of the given route. The route ID has to be specified in the parameter list. The list of segments shows what bus stops it combines (source and target). Also, the geometry of that segment is returned. The list of segments is not sorted.

5.3 Transport Network Information

A function group that returns information about transport network objects that are time-dependent: trips, schedules, etc. The following functions described below belong to this group.

MN_Stop_seq_in_Trip – this function returns the sequence numbers of all bus stops for a given route, trip or just a sequence number of a specific bus stop in a trip. We can specify *route_id*, *trip_id* and *stop_id* or *route_id* and *trip_id*, or just *route_id* in the list of parameters. The sequence numbers in the trip will be returned of a specific bus stop or of bus stops of the given trip or route, respectively, to the number of parameters we specified. The stop, network, route, and trip IDs will be returned along with the sequence of that stop in the trip.

MN_Stop_list_of_Trip – this function returns a list of bus stops of trips for a given route. We can specify *route_id*, *trip_id* as parameters or only the *route_id*. The list of bus stops in the trip is returned in a single text record field. Commas separate the bus stops from one another in the record. The network, route, and trip IDs are returned along with the list of bus stops.

MN_Stop_list_of_Route – this function returns a list of bus stops of distinct trips for a given route. We have to specify *route_id* in the list of parameters. The list of bus stops in the trip is returned in a single text record field. Commas separate the bus stops from one another in the record. The network and route IDs are returned along with the list of bus stops.

MN_R_ST_list – this function returns a name list on of bus stops of distinct trips for a given route. We have to specify *route_id* in the list of parameters. The function returns the name list of bus stops in a single text record field. The list is sorted. Dashes separate the names from one another in the record. The *route_id* is returned along with the list.

MN_R_ST_Time_list – this function returns a list of departure times of distinct routes for a given bus stop. We have to specify the bus *stop_id* in the list of parameters. The bus stop name, *route_id*, and route name are returned along with the list.

5.4 Functions for the Shortest Path Computation

A function group that is important for computations of intermediate results of different applications of Dijkstra algorithm in finding the shortest path, the isochrones, etc. The functions described below are a part of this group.

MN_ST_NearestST_list – this function returns a list of the nearest bus stops that are far from the given bus stop no more than the specified distance or no more than 1 km if the distance is not specified. We can specify the *stop_id*, the distance in kilometers or only the *stop_id*. The function returns 2 lists of textual type, where commas separates stop IDs and distances to the stops. The *stop_id* we provided is returned along with the lists. The distance between stops is calculated as segments connecting the stops.

MN_R_Closest_stops – this function returns full information about valid routes to the closest stops at a given date. We specify *stop_id*, a date and an optional *service_ON* parameter. The optional parameter defines whether service operation period will be taken into consideration or not. The function returns the following type of records: *stop_source* int, *stop_target* int, *edge_id* int, *bn_node_source* int, *bn_node_target* int, *route_id* int, *service_id* int, *link* Boolean. The function is applied to the transport network.

MN_ClosestPoint – this function returns coordinates the shape of text of the given point on the closest street edge, geometry and ID of that edge. We specify *stop_id* as a parameter.

MN_SplitEdge – this function splits a street edge into 2 fragments through a given point of a bus stop. As a result, geometries of the new edges with the total length, equal to the given edge, are returned.

5.5 Example

For clarity, we take the function **MN_Stop_seq_in_Trip** that returns stop sequence numbers in the trip and explains its usage with different parameters in detail. We test the function in PostgreSQL.

The function has the following syntax:

```
MN_Stop_seq_in_Trip(<route_id>,<trip_id>,<stop_id>)
MN_Stop_seq_in_Trip(<route_id>,<trip_id>)
MN_Stop_seq_in_Trip(<route_id>)
```

As you can see, this function has three parameters:

Term	Type	Definition
route_id	int	The route id specifies the route that holds all trips
trip_id	int	The trip id specifies a trip
stop_id	int	The stop id specifies a stop sequence number in the trip

We can specify all of them; *trip_id* with *stop_id* or only single *stop_id*. Some query examples:

```
SELECT * FROM MN_Stop_seq_in_Trip(83,3071,99);
SELECT * FROM MN_Stop_seq_in_Trip(83) WHERE trip_id=3071 AND
stop_id=99;
SELECT * FROM MN_Stop_seq_in_Trip(83,3071);
```

In Table 5.2, we present the result of query that returns the first five stops sequence numbers of *route_id* $= 21$.

Table 5.2. The result of query with the function **MN_Stop_seq_in_Trip**.

```
SELECT * FROM MN_Stop_seq_in_Trip(21) LIMIT 5;
```

net_id	route_id	trip_id	stop_id	value
1	21	4039	263	0
1	21	4039	264	1
1	21	4039	849	2
1	21	4039	266	3
1	21	4039	847	4

(5 rows)

The rest functions can be found in the Appendix.

Chapter 6

Evaluation

Oracle VM VirtualBox with 2 CPU and 3GB of RAM was used to develop and test the multimodal network on the Linux Ubuntu 14.04 OS and PostgreSQL 9.4 database. For testing purposes, Merano–Bolzano pedestrian and transport networks were uploaded to the model. These networks correspond to medium-sized network in Italy. As mentioned before, development of the multimodal network and loading data can be divided into independent stages. Therefore, model testing was run in stages.

Data Loading of Pedestrian Network. After loading the Merano–Bolzano street data into the model tables, we have obtained:

- 9658 street nodes
- 11532 street edges

The process takes approximately 10 seconds together with extraction of the relevant area from the downloaded file and parsing. We exclude the time to download the pedestrian network from Internet.

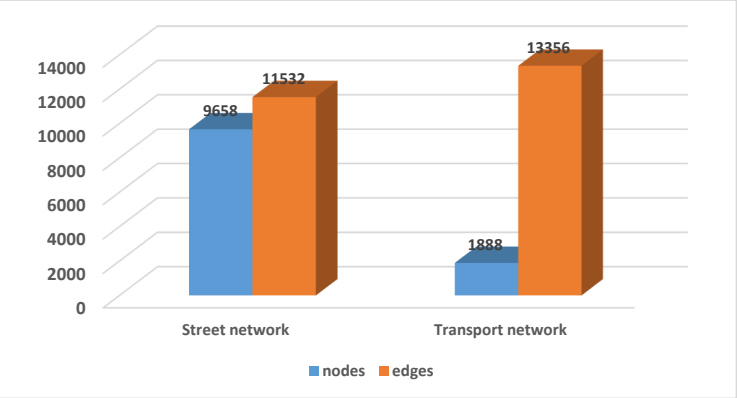

Figure 6.1. Distribution of Merano–Bolzano multimodal network nodes and edges by networks.

Data Loading of Transport Network. The Merano–Bolzano timetable data were parsed and loaded to the model tables:

- 887 stops
- 7560 trips
- 255286 stop times
- 1888 bus network nodes
- 13356 bus network edges

Stop_times file parsing took most time in this process. It took approximately 23 seconds.

Linking of the Pedestrian and Transport Networks. In this process, we dealt with a challenge to link 9658 street nodes and 11532 edges of the pedestrian network with 1888 bus nodes and 13356 edges of the transport network. There arose difficulties because linking had to be done based on geometries. The connection points had to be projections to the closest streets. Instead of scanning all streets for each stop, we drew a rectangle (bounding box) 1 km wide around each stop. Thus, only streets that intersect with rectangles were scanned. Indices and geometry columns were used to check the intersection of streets and bounding box geometries.

After optimizations of computations, we present the results of street splitting process:

- 11532 street edges and 1888 bus nodes were linked;
- 563 street edges were split through 876 new nodes;
- After splitting 1441 new street edges were inserted;
- In total, the splitting process lasted approximately 2 seconds.

Conclusions and future perspectives. To test the effectiveness of the model, we selected the synthetic tests: the same networks were considered as distinct ones and were imported into the model several times. The tests have showed that:

- Time of network import and linking is almost linearly dependent only upon the number of nodes and edges of networks we have imported;
- Only slightly they are dependent on the number of already imported networks in the model (see Figs. 6.2–6.4).

The analysis of pedestrian and transport networks has showed that the pedestrian network has on average about 10 times more nodes than the transport network in the same region (see Fig. 6.1). Moreover, the pedestrian network does not use timetables. To sum up, we can optimize the query time by putting the preprocessing effort to the pedestrian network.

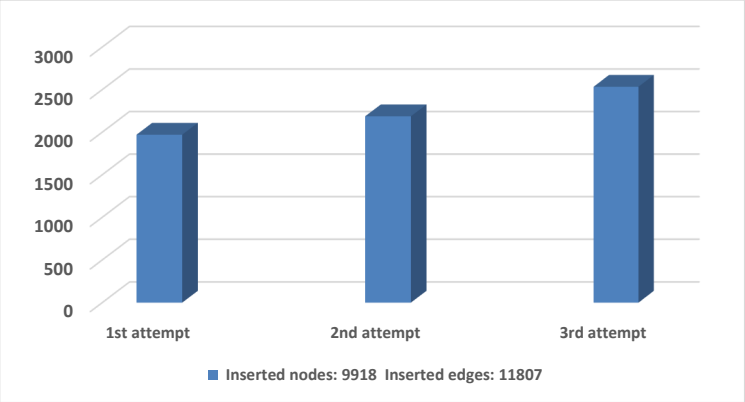

Figure 6.2. Import time of the pedestrian network in ms.

Figure 6.3. Import time of the transport network in ms.

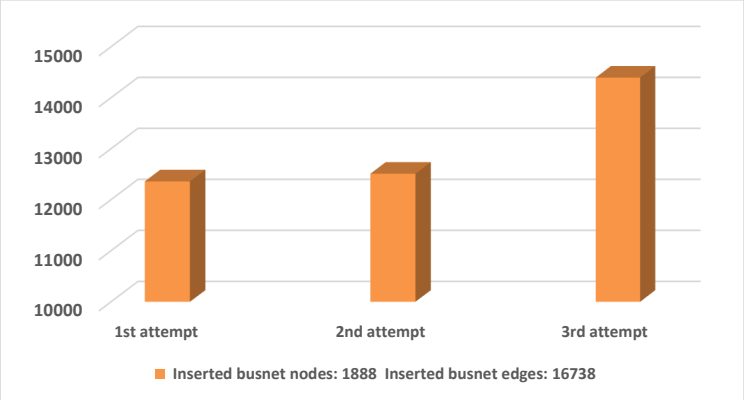

Figure 6.4. Time of the linking networks in ms.

Chapter 7

Conclusion and Future Work

The last section is dedicated to the final inferences about our proposed model, the advantages and disadvantages against a similar type of other models as well as future perspective of this model for further development and exploitation. The biggest benefit of this model is the solution to store all the pedestrian networks in the same tables. We apply the same solution to the transport networks. It may seem that it can affect data search, however necessary indices on data, clustering and other database techniques makes the search fast and efficient.

Loading of the new networks into the model is organized using database tools. Only some parts like downloading from Internet, extracting region from a map or file conversion from one format to another use third-party software. The database tools used for loading data ensure the speed and efficiency of this process. The short loading time into the model enables these and other works related to the model to be carried out in the interactive mode.

A set of implemented functions makes it easier to work with a model for a wider range of users, who do not need to know the internal structure of the model.

Obviously, the model has some drawbacks that can be easily improved in the future. First of all, there is poor information about the pedestrian network. The original source of the roads (OSM files) contains more information such as street names, direction (one-way, bidirectional), valid speed of that street, etc. This information is refused in our model.

There is also a proposal to use precomputed data, i.e. precompute the shortest distances between stops for the pedestrian network. Since we can add more than one network in the model, it would be good to compute distances not among all stops, but only those, which are far from each other not more than the given distance (for example, 5 km).

It is necessary to expand the multimodal DB API (increase a variety of functions to work with the model) in order to make our model simpler and

more accessible for a greater number of users. In future, it would be useful to offer this model to other OS platforms such as Windows and other DBMS such as Oracle.

Appendix

Comprehensive definitions of query services together with parameters and examples of queries are described below.

Function MN_Stop_list_of_Trip

This is a function that returns the stop list of a trip.

Syntax

```
MN_Stop_list_of_trip(<route_id>,<trip_id>)
MN_Stop_list_of_trip(<route_id>)
```

Parameters

Term	Type	Definition
route_id	int	The route id specifies the route that holds all trips
trip_id	int	The trip id specifies a trip that holds a stop list

Return Value

A record set of *network_id* int, *route_id* int, *trip_id* int, *stop_list* text.

Example

The query below shows the usage of the function with different parameters:

```
SELECT * FROM MN_Stop_list_of_trip(83,3071);
```

Table A.1. The result of query with the function **MN_Stop_list_of_Trip**.

```
SELECT * FROM MN_Stop_list_of_trip(21) LIMIT 5;
```

net_id	route_id	trip_id	list
1	21	4039	263, 264, 849, 266, 847, 269, 270, 261, 260, 271, 272, 473
1	21	4040	473, 272, 326, 259, 262, 268, 267, 848, 265, 850, 263
1	21	4041	473, 272, 326, 259, 262, 268, 267, 848, 265, 850, 263
1	21	4042	263, 264, 849, 266, 847, 269, 270, 261, 260, 271, 272, 473
1	21	4043	473, 272, 326, 259, 262, 268, 267, 848, 265, 850, 263

(5 rows)

Function MN_Stop_list_of_Route

This is a function that returns the different stop lists of trips of the route.

Syntax

```
MN_Stop_list_of_route(<route_id>)
```

Parameters

Term	Type	Definition
route_id	int	The route id specifies the route that holds different stop lists of trips

Return Value

A record set of *network_id* int, *route_id* int, *stop_list* text.

Example

The following query shows the usage of the function:

```
SELECT * FROM MN_Stop_list_of_route(21);
```

Table A.2. The result of query with the function **MN_Stop_list_of_Route**.

```
SELECT * FROM MN_Stop_list_of_route(21);
```

net_id	route_id	list
1	21	263, 264, 849, 266, 847, 269, 270
1	21	473, 272, 326, 259, 262, 268, 267, 848, 265, 850, 263
1	21	263, 264, 849, 266, 847, 269, 270, 261, 260, 271, 272, 473

(3 rows)

Functions MN_Route_names & MN_Net_routes

These are functions that return a list of route names.

Syntax

```
MN_Route_names(<route_id>)
MN_Route_names()
MN_Route_names(<route_name_context>)
MN_Net_Routes(<net_i>)
```

Parameters

Term	Type	Definition
route_id	int	The route id specifies the route, whose name should be returned (if route_id is unspecified, all route names will be returned)
route_name_context	text	Context of route name

Return Value

A record set of *network_id* int, *route_id* int, *short_name* text, *long_name*.

Examples

The following queries shows the usage of the function with different parameters:

```
SELECT * FROM MN_Route_names(83);
SELECT * FROM MN_Route_names() WHERE route_id=83;
SELECT * FROM MN_Route_names('4');
SELECT * FROM MN_Net_routes(2);
```

Table A.3. The result of query with the function **MN_Route_names**.

```
SELECT * FROM MN_Route_names(21);
```

net_id	route_id	short_name	long_name
1	21	222 ME	222

(1 row)

Table A.4. The result of query with the function **MN_Net_routes**.

```
SELECT * FROM MN_Net_routes(1) LIMIT 4;
```

net_id	route_id	short_name	long_name
1	1	1 ME	1
1	2	2 ME	2
1	3	3 ME	3
1	4	4 ME	4

(4 rows)

Function MN_R_segments

This is a function that returns the list of route segments.

Syntax

```
MN_R_segments(<route_id>)
```

Parameters

Term	Type	Definition
route_id	int	The route id specifies the route, whose segments should be returned

Return Value

A record set of *route_id* int, *stop_source* int, *stop_target* int, *segment_geometry* geometry.

Examples

The following queries shows the usage of the function:

```
SELECT * FROM MN_R_segments(83);
SELECT route_id, stop_source, stop_target, ST_AsText(segment_geometry)
FROM MN_R_segments(83);
```

Table A.5. The result of query with the function **MN_R_segments**.

```
SELECT * FROM MN_R_segments(21) LIMIT 5;
```

route_id	stop_source	stop_target	segment_geometry
21	259	262	0102000020E610000002000000A9FB00A...
21	260	271	0102000020E610000002000000DE8D058...
21	261	260	0102000020E6100000020000000875914...
21	262	268	0102000020E610000002000000F2B0506...
21	263	264	0102000020E6100000200000018EB1B9...

(5 rows)

Table A.6. The result of query with the function **MN_R_segments**.

```
SELECT route_id, stop_source, stop_target, ST_AsText(segment_geometry)
                    FROM MN_R_segments(21) LIMIT 5;
```

route_id	stop_source	stop_target	st_astext
21	259	262	LINESTRING(11.158 46.686,11.155 46.689)
21	260	271	LINESTRING(11.159 46.686,11.161 46.682)
21	261	260	LINESTRING(11.156 46.687,11.158 46.686)
21	262	268	LINESTRING(11.155 46.689,11.154 46.691)
21	263	264	LINESTRING(11.157 46.702,11.154 46.700)

(5 rows)

Function MN_R_ST_Time_list

This is a function that returns the route departure times in the bus stop.

Syntax

```
MN_R_ST_Time_list(<stop_id>)
```

Parameters

Term	Type	Definition
stop_id	int	The stop that holds departure times of the routes

Return Value

A record set of *stop_id* int, *stop_name* text, *route_id* int, *route_long_name* text, *times_list* text.

Example

The following query shows the usage of the function:

```
SELECT stop_id, stop_name, route_id, route_long_name,
left(times_list,50) FROM MN_R_ST_Time_list(1) LIMIT 5;
```

Table A.7. The result of query with the function **MN_R_ST_Time_list**.

```
SELECT stop_id, stop_name, route_id, route_long_name, left(times_list,50)
                    FROM MN_R_ST_Time_list(1) LIMIT 5;
```

stop_id	stop_name	route_id	route_long_name
1	Stazione Merano - Bhf Meran	20	221
1	Stazione Merano - Bhf Meran	40	999
1	Stazione Merano - Bhf Meran	1	1
1	Stazione Merano - Bhf Meran	11	146
1	Stazione Merano - Bhf Meran	4	4

stop_id	left(times_list,50)
1	6:20, 6:20, 6:20, 6:55, 6:55, 6:55, 6:56, 6:56, 6:
1	22:53, 23:45, 24:6, 24:45, 24:58, 25:45, 25:58, 26
1	6:19, 6:19, 6:19, 6:44, 6:44, 6:44, 6:44, 7:4, 7:4
1	20:0, 20:0, 20:0, 20:20, 20:20, 20:20, 20:22, 20:2
1	5:49, 5:49, 5:49, 5:52, 5:52, 5:52, 6:21, 6:21, 6:

(5 rows)

Function MN_R_ST_list

This is a function that returns the name list of stops of different trips.

Syntax

```
MN_ST_of_R(<route_id>)
```

Parameters

Term	Type	Definition
route_id	int	The route id specifies route that holds the list of stops

Return Value

A record set of *stop_list*.

Example

The following query shows the usage of the function:

```
SELECT route_id, left(stops_list,80) FROM MN_R_ST_list(21);
```

Table A.8. The result of query with the function **MN_R_ST_list**.

```
SELECT route_id, left(stops_list,80) FROM MN_R_ST_list(21);
```

route_id	left(stops_list,80)
21	Tirolo Croce - Tiroler Kreuz - - Lechner - - Pamer Kreuz - Funivia Muta - Seilba
21	Monte S. Benedetto Seggiovia - Segenbühe - Salita Tirolo - Tirolersteig - Lido -
21	Tirolo Croce - Tiroler Kreuz - - Lechner - - Pamer Kreuz - Funivia Muta - Seilba

(3 rows)

Function MN_ST_list

This is a function that returns the stop list of a specific transport network.

Syntax

MN_ST_list(<net_id>)

Parameters

Term	Type	Definition
net_id	int	The transport network that holds the list of stops

Return Value

A record set of *stop_id* int, *stop_name* text.

Examples

The following queries shows the usage of the function with different parameters:

```
SELECT * FROM MN_ST_list(2);
SELECT * FROM MN_ST_list(2)
WHERE stop_name like '%Wein%';
SELECT * FROM MN_ST_list(1)
WHERE stop_id = 517;
```

Table A.9. The result of query with the function **MN_ST_list**.

```
SELECT * FROM MN_ST_list(1) LIMIT 3;
```

stop_id	stop_name
1	Stazione Merano - Bhf Meran
2	Via delle Corse - Rennweg
3	Ospedale - Krankenhaus

(3 rows)

Function MN_ST_of_R

This is a function that returns the stops of a specific route.

Syntax

```
MN_ST_of_R(<route_id>)
```

Parameters

Term	Type	Definition
route_id	int	The route id specifies route that holds all the stops

Return Value

A record set of *stop_id* int.

Example

The following query shows the usage of the function:

```
SELECT * FROM MN_ST_of_R(21) LIMIT 5;
```

Table A.10. The result of query with the function **MN_ST_of_R**.

```
SELECT * FROM MN_ST_of_R(21) LIMIT 3;
```

mn_st_of_r
264
259
269
(3 rows)

Function MN_ST_NearestST_list

This is a function that returns the list of the nearest stops from a selected bus stop.

Syntax

```
MN_ST_NearestST_list(<stop_id>)
MN_ST_NearestST_list(<stop_id>,<distance_in_km>)
```

Parameters

Term	Type	Definition
stop_id	int	The selected stop id
distance_in_km	double precision	Distance in km

Return Value

A record set of *stop_id* int, *stop_list* text, *dist_list* text.

- **Important**

If *distance_in_km* is not specified, the function takes the distance by default (≈ 150 meters).

Examples

The following queries shows the usage of the function with different parameters:

```
SELECT * FROM MN_ST_NearestST_list(732)
SELECT * FROM MN_ST_NearestST_list(732,0.5);
```

Table A.11. The result of query with the function **MN_ST_NearestST_list**.

```
SELECT * FROM MN_ST_NearestST_list(1);
```

stop_id	stop_list	dist_list
1	1; 142; 143; 171; 368; 775	0.00; 0.11; 0.12; 0.08; 0.11; 0.13
(1 row)		

```
SELECT * FROM MN_ST_NearestST_list(1,0.2);
```

stop_id	stop_list	dist_list
1	1; 142; 143; 145; 171; 368; 775	0.00; 0.11; 0.12; 0.20; 0.08; 0.11; 0.13
(1 row)		

Function MN_R_Closest_stops

This is a function that returns full information about valid routes to the closest stops for a given date.

Syntax

```
MN_R_Closest_stops(<stop_id>,<date>)
MN_R_Closest_stops(<stop_id>,<date>,<service_ON|OFF>)
```

Parameters

Term	Type	Definition
stop_id	int	The given bus stop id from where we start search of neighbour stops
date	text	The given date to check valid services on that day
service_ON|OFF	Boolean	The service usage is on/off

Return Value

A record set of *stop_source* int, *stop_target* int, *edge_id* int, *bn_node_source* int, *bn_node_target* int, *route_id* int, *service_id* int, *link* Boolean>

- **Important**

If the service parameter is FALSE, the valid date of service will be disregarded.

Examples

The following queries show the usage of the function with different parameters:

```
SELECT * FROM MN_R_Closest_stops(1,'20160415');
SELECT * FROM MN_R_Closest_stops(1,'20160415',FALSE);
```

Table A.12. The result of query with the function **MN_ST_NearestST_list**.

```
SELECT * FROM MN_R_Closest_stops(1,'20160415',FALSE);
```

stop_source	stop_target	edge_id	bn_node_source
1	11	2185	10
1	11	8127	10
1	11	8587	10
1	11	6710	10
1	12	2455	7
1	12	5009	7
bn_node_target	**route**	**service**	**link**
48	96	723	f
48	96	724	f
48	96	725	f
48	96	727	f
56	85	722	f
56	85	723	f

(6 rows)

Function MN_ClosestPoint

This is a function that returns a projection on the closest street edge to a bus stop.

Syntax

```
MN_ClosestPoint(<stop_id>)
```

Parameters

Term	Type	Definition
stop_id	int	The given bus stop id, whom we search for the closest edge

Return Value

A record set of *point* text, *edge* text.

Example

The query below shows the usage of the function with different parameters:

```
SELECT * FROM MN_ClosestPoint(1);
```

Table A.13. The result of query with the function **MN_ClosestPoint**.

```
SELECT * FROM MN_ClosestPoint(45);
```

point	edge_id	edge
POINT(11.193 46.608)	12814	LINESTRING(11.193 46.608,11.193 46.608,...)

(1 row)

Function MN_SplitEdge

This is a function that splits a street edge into 2 fragments through a given point for a bus stop.

Syntax

```
MN_SplitEdge(<stop_id>,<point>)
```

Parameters

Term	Type	Definition
stop_id	int	The given bus stop id
point	text	Point coordinates on the edge, which we want to split

Return Value

A record set of *point* text, *edge1* text, *edge2* text.

Example

The query below shows the usage of the function with different parameters:

```
SELECT * FROM MN_SplitEdge(1,'POINT(11.150 46.673)');
```

Table A.14. The result of query with the function **MN_SplitEdge**.

```
SELECT * FROM MN_SplitEdge(45,'POINT(11.193 46.608)');
```

point	edge, edge1, edge2
POINT(11.198 46.608)	LINESTRING(11.174 46.696,11.174 46.696,11.174 45.696,...)
	POINT(11.174 46.696)
	LINESTRING(11.174 46.696,11.174 46.696,...)

(1 row)

Bibliography

[1] J. Arz, D. Luxen, and P. Sanders. Transit node routing reconsidered. In V. Bonifaci, C. Demetrescu, and A. Marchetti-Spaccamela, editors, *Experimental Algorithms 12th International Symposium, SEA 2013 Rome, Italy, June 5–7, 2013. Proceedings*, Lect. Notes Comput. Sci., Vol. 7933, pages 55–66. Springer, Berlin, Heidelberg, 2013.

[2] H. Bast, D. Delling, A.V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R.F. Werneck. Route planning on transportation network. Technical Report MSR-TR-2014-4, Microsoft Research, Microsoft Corporation, Redmond, WA, 2014.

[3] J. Booth, P. Sistia, O. Wolfson, and I.F. Cruz. A data model for trip planning in multimodal transportation system. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, Saint-Petersburg, Russia, March 23–26, 2009*, pages 994–1005, 2009.

[4] D. Delling, T. Pajor, and D. Wagner. Engineering time-expanded graphs for faster timetable information. In *Robust and Online Large-Scale Optimization*, Lect. Notes Comput. Sci., Vol. 5868, pages 182–206. Springer, Berlin, Heidelberg, 2009.

[5] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*, Lect. Notes Comput. Sci., Vol. 5515, pages 117–139. Springer, Berlin, Heidelberg, 2009.

[6] W. Edsger. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[7] R. Geisberger. *Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks*. PhD thesis, Institut für Theoretische Informatik, Universität Karlsruhe, 2008.

[8] R.F. Mahrous. Multimodal transportation systems: Modelling challenges. Master thesis, University of Twente, 2012.

[9] R.H. Möhring, H. Schillingand, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up dijkstra's algorithm. In S.E. Nikoletseas, editor, *Experimental and Efficient Algorithms. 4th International Workshop, WEA 2005, Santorini Island, Greece, May 10–13, 2005. Proceedings*, Lect. Notes Comput. Sci., Vol. 3503, pages 189–202. Springer, Berlin, Heidelberg, 2005.

[10] T. Pajor. *Multi-Modal Route Planning*. PhD thesis, Institut für Theoretische Informatik, Universität Karlsruhe, 2009.

[11] E. Pyrga and C. Zaroliagis. Efficient models for timetable information in public transportation systems. *Journal of Experimental Algorithmics*, 12, Article No. 2.4, 2008.

[12] A.L. Rosenberg and L.S. Heath. *Graph Separators, with Applications*. Front. Comput. Sci. Kluwer Academic, New York, 2002.

[13] W. Roush. Welcome to Google transit: How (and why) the search giant is remapping public transportation. *Community Transportation*, 3:20–29, 2012.

[14] J. Zhang, F. Liao, T. Arentze, and H. Timmermans. A multimodal transport network model for advanced traveler information systems. *Procedia – Social and Behavioral Sciences*, 20:313–322, 2011.

[15] OpenStreetMap. `http://www.openstreetmap.org`.

[16] Osm2po is both, a converter and a routing engine. `http://osm2po.de`.

[17] PostgreSQL 9.4.8 Documentation. `https://www.postgresql.org/docs/9.4/static/tutorial-sql.html`.

[18] The General Transit Feed Specification. `https://developers.google.com/transit/gtfs/`.

[19] Verband Deutscher Verkehrsunternehmen. `http://www.vdv.de/oepnv-datenmodell.aspx`.