London Safe Travel

Dario Pagani 585281
Federico Frati 596237
Ricky Marinsalda 585094

January 25, 2023

Contents

I Introduction **3**

II Feasibility **5**

1 Data sources **6**
1.1 Transport for London 6
1.2 OpenStreetMap 6

III Design **8**

2 Main Actors **9**

3 Functional requirements **10**
3.1 Client 10
3.2 TIMS 10
3.3 Statician 11

4 Not functional requirements **12**

5 CAP Theorem issue **13**

6 Use cases **14**

7 Databases design **16**

8 Data model **18**
8.1 Document database 18
8.2 Graph database 20

9 Distributed database design **23**
9.1 Replica set 23
9.2 Replica Configuration 24
9.3 Replica crash 25

10 Overall platform architecture **26**

11 Technologies used **27**

IV Implementation **28**

12 Java application **29**
12.1 Source tree 29
12.1.1 Packages 31
12.1.2 Java model classes 31

12.2 Considerations on geographical data . . . 34
12.3 Maven build system . . . 38

13 DocumentDB CRUD operations **40**
13.1 Disruption: Create and Update . . . 40
13.2 Disruption: Read . . . 40
13.3 Point of interest: Create . . . 40
13.4 Point of Interest: Read . . . 41

14 Graph Database CRUD operations **42**
14.1 Bootstrap . . . 42
14.2 Point: Create . . . 42
14.3 Way: Create . . . 42
14.4 Disruption: Create and Update . . . 43
14.5 Disruption: Read . . . 44
14.6 Disruption: Delete . . . 45

15 Queries on graph database **46**
15.1 Routing . . . 46
15.2 Point finding . . . 51
15.3 Create and update disruptions . . . 52

16 Queries on Document database **53**
16.1 POIs in a certain area . . . 53
16.2 The heatmap . . . 54
16.3 The most common disruptions . . . 56
16.4 Time series . . . 57
16.5 Find a place . . . 61

17 Neo4j indexes **62**
17.1 Geo-spatial index . . . 62

18 MongoDB indexes **65**
18.1 Point of Interest . . . 65

19 TIMS client **68**
19.1 Implementation . . . 68
19.2 Configuration . . . 68

V Conclusion **70**
19.3 The Benefits of Using Graph and Document Databases in Transportation Systems 71
19.4 Possible expansions in the future . . . 71
19.5 Possible query on public trasportations . . . 71

VI Manual **72**

A User **73**
A.1 Point of interests . . . 74
A.2 Disruptions . . . 75
A.3 Routing . . . 76
A.4 Search . . . 78

B Statistician **79**
B.1 Heat-map . . . 79
B.2 Disruptions' time series . . . 80
B.3 Common disruptions in an area . . . 80

Part I

Introduction

Every day millions of people find themselves in the troublesome task of computing a route to their desired destination. We'll provide directions for their journey with state of the art routing algorithms in a safe way. The service will provide the following transportation modes when querying a route: car, foot and bicycle. We'll also provide a compact and easy way to view timetables and public transport's routes.

Problem Navigating a city like London is a complicated matter even during normal times, let alone during rush hour. Due to the convoluted nature of London's road network accidents and disruptions are common occurrences that afflict all road users during their journeys. People who use public transportation to travel around the city could be interested in knowing which are the most congested lines in advance in order to avoid delays and other kinds of interference.

Objective We'll provide analytic functionalities to find hotspots in the road network graph, that is to collect information about all users' journeys and visualize the most congested routes/graph's regions. Other information about public transportation's issues will help users to locate and avoid the most critical parts of the transportation network.

Part II

Feasibility

Chapter 1

Data sources

We're combining data from two organizations:

- Transport for London TIMS
- OpenStreetMap

Introduction At the very beginning, we made a feasibility study of the project idea, in order to identify both the pros and the critical aspects of the application, considering the time and technical constraints at the time and thinking how to possibly improve our work in future.

1.1 Transport for London

Scraping Transport for London provides data at regular intervals – every five minutes – and in a standardized format; this kind of data, especially when scrapped during an adequate period of time can provide a good amount of information to be analyzed by our application. Those analyses become of particular interest when we consider that Transport for London provides data with a great amount of detail, including things such as minor collisions and broken traffic lights. In our particular case we collected data for circa a month using *Transport for London*'s JSON API with a simple shell script that was being executed automatically by a *SystemD* timer every ten minutes.

Format An example

Schema The main challenge was to find a "schema" for the document database to store terminated disruption to make possible to execute fast analytics on them.

Sources It is possible to access the documentation of the *API* from here: `https://api-portal.tfl.gov.uk/`

1.2 OpenStreetMap

Scraping Since the map's data would be used only for routing purposes, we don't need all the information provided by OpenStreetMap; so, to reduce the dimension of the data and maintain only the useful information, we decided to do some pruning of the network graph, in particular we removed useless nodes such as buildings, waterways, trees, parks and so on.

Schema The main challenge was to find a good representation of OpenStreetMap's data for our chosen graph database (Neo4j) because it has to represents facts like one ways streets, access restrictions and the elements' geographical positions; we iterated over several possible schemas for the graph database and over many different ways to import the raw data into it.

Format Data is provided in an *XML* format, after some processing we obtained two *CSV*s one containing all the intersection in the network and the other all the highway's stretches, that is the edges between intersections, in the network.

```
1 id:ID,latitude:double,longitude:double,coord:point{crs:WGS-84},:LABEL
2 78112,51.526976,-0.1457924,"{latitude:51.526976, longitude:-0.1457924}","Point"
3 99878,51.524358,-0.1529847,"{latitude:51.524358, longitude:-0.1529847}","Point"
4 99879,51.5248246,-0.1532934,"{latitude:51.5248246, longitude:-0.1532934}","
    ↪ Point"
5 99880,51.5250847,-0.1535802,"{latitude:51.5250847, longitude:-0.1535802}","
    ↪ Point"
```

Figure 1.1: Example of intersections

```
1 p1:START_ID,p2:END_ID,id:long,name,class,maxspeed:double,crossTimeFoot:double,
    ↪ crossTimeBicycle:double,crossTimeMotorVehicle:double,:TYPE
2 196101,2121445348,74,Ballards Lane,primary
    ↪ ,13.4112,14.55373983562014,3.638434958905035,1.3564922861686513,CONNECTS
3 2121445348,196101,74,Ballards Lane,primary
    ↪ ,13.4112,14.55373983562014,3.638434958905035,1.3564922861686513,CONNECTS
4 196055,1030634587,75,High Road,primary
    ↪ ,13.4112,43.757620141633296,10.939405035408324,4.078461952298211,
    ↪ CONNECTS
5 1030634587,196055,75,High Road,primary
    ↪ ,13.4112,43.757620141633296,10.939405035408324,4.078461952298211,
    ↪ CONNECTS
```

Figure 1.2: Example of intersections

Sources We imported the map's data from an *XML* file provided by Geofrabik from here: `https://download.geofabrik.de/europe/great-britain/england/greater-london.html`

Part III

Design

Chapter 2

Main Actors

The application has three main actors:

- Client
- TIMS
- Statician

Client this user is able to view map and run searches on it and he's also able to ask for directions between two points on the map. The user don't need to be registered to use the application. With correct use of the app, the client can avoid inconvenience during their journey in the city of London, completing his visit in safety and minimizing delays.

TIMS this user is able to perform CRUD operations on the disruptions. TIMS, in pratics, is an API provided by Transport For London.

Statician this user is able to perform analytics on the data generated by TIMS. The statician can access his panel inserting the correct username and password. By acceding this panel he can view some different graphs that are useful to make statistics about security when travelling in London. Using these results, the statistician can identify critical issues in the city of London and suggest to the authorities any changes to traffic or simply indicate which areas are the most critical from this point of view.

Chapter 3

Functional requirements

The application will allow its users to perform the following tasks:

3.1 Client

- **View the map**
- **Move the map in a direction**, with the typical drag 'n drop
- **Increase or decrease the map magnifying**, with either buttons or the mouse's wheel
- **Search for a specific place**, the following method are supported:
  - With an address
  - With a POI's name
  - With WGS84 geographical coordinates
- **Ask for directions** between two intersections on the map
  - Via car
  - Via bicycle
  - On foot
- **Calculate the estimate time of travel**
- **View the path** on the map
- **View active disruptions' and POIs' informations**
- **View a suggestion list of POIs**, based on the string the user insert in the appropriate area
- **Refresh active disruptions' data** clicking on the refresh button
- **Choose if to consider or not disruptions** during the route calculation

3.2 TIMS

- **Create/Update a disruption**
- **Close a disruption**

3.3 Statician

- **Access the analytics tools**
- **Display the disruptions' heatmap**, chosen a certaine class
- **Choose the precision of the hehatmap between three grades:**
  - High
  - Medium
  - Low
- **Visualize a graph about hourly statistics chosen a class of disruption**
- **Visualize a table that contains the most common disruption for each severity considering the area actually displayed on the screen**
- **Update the table**

Chapter 4

Not functional requirements

- To solve a routing problem in an acceptable amount of time
- A cache for queries to have a better response time for frequent queries
- To find a good approximation of the optimal route, finding a good balance between the optimal result and execution time of the procedure on the server
- Respect the access constraints on the graph, such as oneway streets and other access restrictions imposed by the highway's authority
- Use a routing algorithm that uses an heuristic function to compute a path, such as *A**, to avoid excessive exploration of the hypothesis space, since those algorithms are always exponential in the graph's dimension
- Define a good heuristic to guide the algorithm on its visit on the graph, considering features such as the road's classification, speed limit, the geographical distance between the vertices.
- To ensure availability of the data and partition tolerance
- Define appropriate requests by the application to minimize the data exchange between the client and the server over the network, in particular to reduce latency of the application when retrieving the *Point of Interest*s
- Define appropriate auxiliary data structure to speed-up the queries on the geographical data
- Define appropriate redundancies to allow the user to quickly access information regarding the currently active disruptions. So we have to avoid to access useless information from the archived data
- Ensure that in the event of a problem in the TIMS' logic or in the *Transport for London*'s *API*s the active disruptions will remain active after their programmed expiration date

Chapter 5

CAP Theorem issue

In order to optimize performance for the anticipated high volume of read operations, it is essential to prioritize both high availability and low latency in the design of this application. Additionally, it is crucial that the system remains functional in the event of a partition.According to the CAP theorem, the design of this application should prioritize **Availability (A)** and **Partition Tolerance (P)** over Consistency (C). This means that the application is more focused on maintaining access to the system and tolerating partitioning rather than ensuring complete consistency of data.

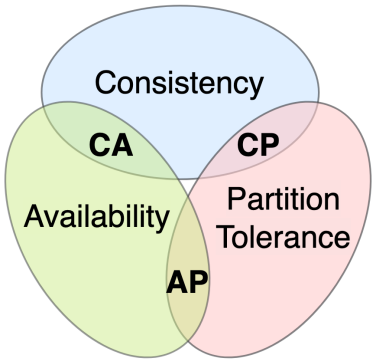


Figure 5.1: CAP theorem Venn diagram

Chapter 6

Use cases

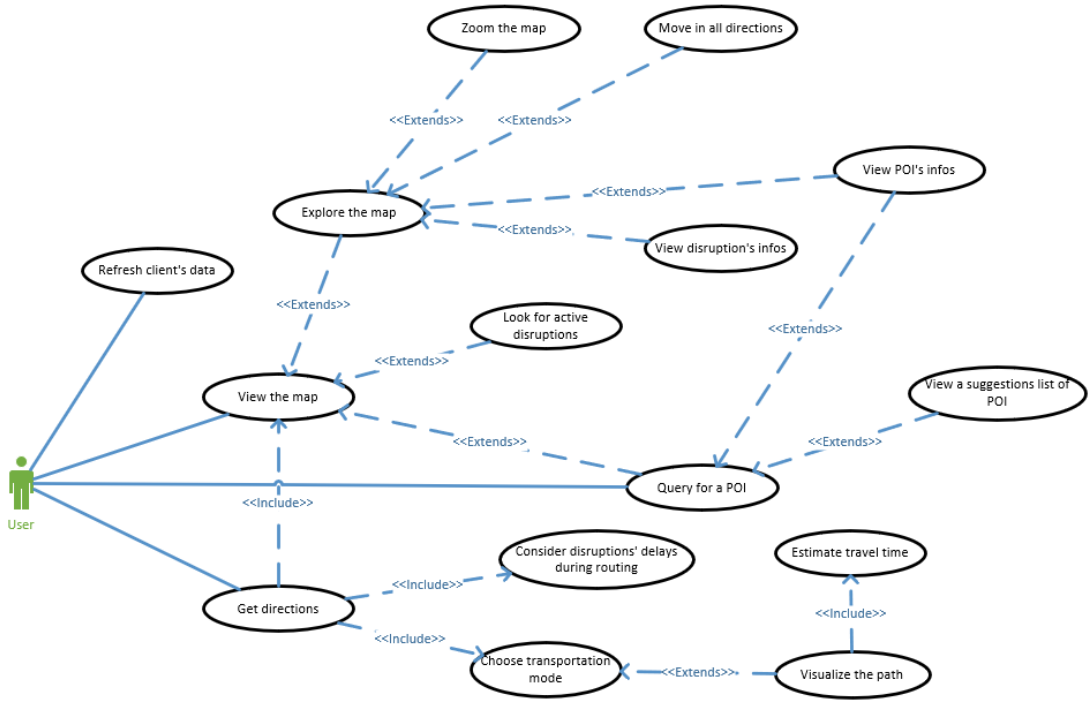
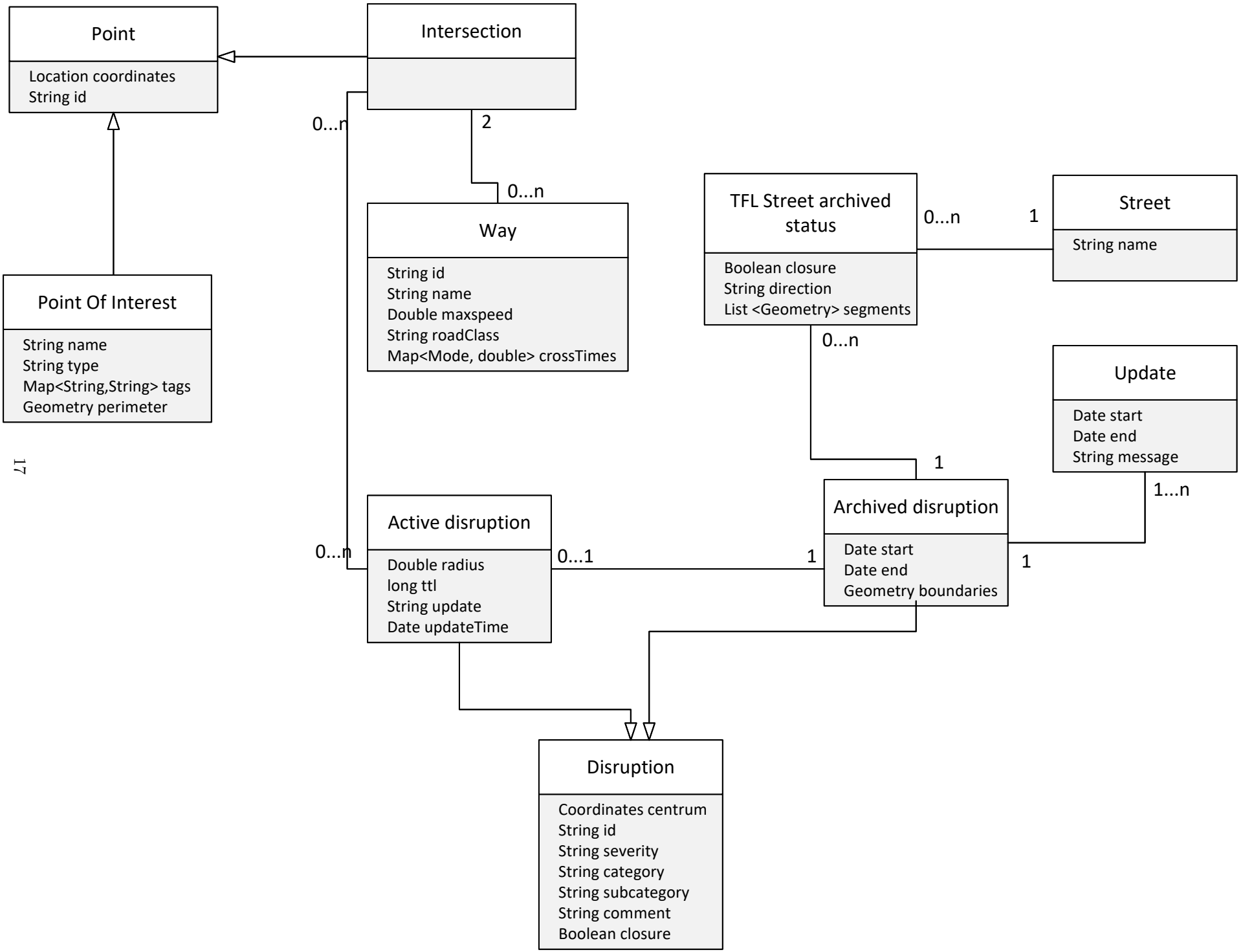


Figure 6.1: Use case diagram: User

Figure 6.2: Use case: TIMS and statician

Chapter 7

Databases design


Point
Location coordinates
String id
Intersection
0...n
2
0...n
Way
String id
String name
Double maxspeed
String roadClass
Map<Mode, double> crossTimes
Point Of Interest
String name
String type
Map<String,String> tags
Geometry perimeter
TFL Street archived status
Boolean closure
String direction
List <Geometry> segments
0...n
1
Street
String name
0...n
1
Update
Date start
Date end
String message
1...n
Active disruption
Double radius
long ttl
String update
Date updateTime
0...n
0...1
1
Archived disruption
Date start
Date end
Geometry boundaries
1
Disruption
Coordinates centrum
String id
String severity
String category
String subcategory
String comment
Boolean closure

Chapter 8

Data model

The data model section includes a description of the document collections, graph nodes and keys that are stored in the database. Two types of DBMSes have been selected for this purpose.

Document DB MongoDB has been chosen as the database management system for the document database part, which stores information about the Point Of Interest and the concluded disruptions.

The decision to use a document database was based on its flexibility and ability to perform complex queries and works as a way to store an history of these informations.

Graph DB To manage the routing, where users can insert two points and the application will compute a travel route between them, a graph database managed using Neo4j has been selected to better support these features.

8.1 Document database

DBMS Our choice for the *database management system* to handle the document database was *MongoDB*, since it is the most popular *DBMS* of its kind and it also provides several functionalities useful for our use case, such as indexes and a powerful query engine.

Collections In MongoDB we created the following collections:

- POIs
- Disruptions

Point of Interest

The POIs collection looks like this:

```
{
  "name": "",
  "coordinates": {
    "type": "Point",
    "coordinates": [-0.1558246, 51.530421]
  },
  "poiID": "388826",
  "tags": {
    "board_type": "wildlife",
    "tourism": "information",
    "information": "board"
  },
  "type": "OSM-POI"
}
```

Where the map *tags* has not a particular structure, but it stores the *OpenStreetMap*'s tag for that particular POI. It might store information such as its website, its opening hours, its physical dimensions, if it is accessible in a wheelchair etc...

Main fields

- `coordinates` a *GeoJson* document of type `Point` representing the geographical location of the object
- `name` the name of the object, if any

Disruption

The disruption collection is organized in the following structure:

```
{
  "boundaries": {
    "type": "Polygon",
    "coordinates": [[
       [-0.2941527354, 51.4913972513], [-0.2941144636, 51.4911846774],
       ..., [-0.2941527354, 51.4913972513]
     ]]
  },
  "category": "Works",
  "coordinates": {
    "type": "Point",
    "coordinates": [-0.27671, 51.489055]
  },
  "end": {"$date": "2023-01-31T23:59:00.000Z"},
  "id": "TIMS-265653",
  "severity": "Moderate",
  "start": {"$date": "2022-03-07T22:00:00.000Z"},
  "streets": [
    {
      "closure": false,
      "direction": "All Directions",
      "name": "[A3000] Wellesley Road (W4)",
      "segments": [
        {
          "type": "LineString",
          "coordinates": [[-0.279577, 51.490059], [-0.279417, 51.490084]]
        },...
      ]
    }
  ],
  "subCategory": "TfL",
  "updates": [
    {
      "end": {"$date": "2022-12-09T14:04:53.000Z"},
      "message": "",
      "start": {"$date": "2022-12-09T13:08:54.000Z"}
    }, ...
  ]
}
```

This representation might allow in the future to store additional optional values for certain kinds of POIs if the need arises, without any compatibility issue for the existing code; for example one might want to store a restaurant's opening hours or the accessibility level for wheelchair users in a certain building.

Main fields

- `boundaries` a *GeoJson* document that could be both of class `Polygon` or `Multipolygon`, it represents the effected area. It might not be present in all documents.
- `coordinates` the main point effected by the disruption, that is where the application is supposed to draw the sign.
- `streets` a list of streets effected by the disruptions and if are closed or not to the traffic
- `updates` an archive of updates that *Transport for London* has published

8.2 Graph database

DBMS Our choice for the *database management system* to handle the graph database was *Neo4j*, since it can easly handle a huge amount of nodes.

Data The graph DB is mainly used to store the road network of London and the current active disruption, this is needed so that the routing algorithm can avoid adding to the frontier nodes in an area affected by a closure or increase the weight of nodes in areas affected by critical disruptions.

Schema For the map side of things we store one class of node and one class of relationship:

- **Intersection** (due to a mistake it is actually called Point in the database) represents the connection between one or more ways, it is a *vertex* of the graph

```
(:Point {
  coord: point({srid:4326, x:-0.1457924, y:51.526976}),
  id: 78112
})
```

  - `coord` Those are the geographical coordinates of the node

- **Connects** represents the connection between two Insersection. It stores informations like its name and the cost of traversing it and eventual access restrictions, like one way streets or motor-only roads

```
[:CONNECTS {
  name: "Ballards Lane",
  maxspeed: 13.411,
  id: 74,
  crossTimeBicycle: 3.638,
  crossTimeFoot: 14.553,
  crossTimeMotorVehicle: Infinity,
  class: "pedestrian",
}]
```

  - `maxspeed` It is the maximum speed on that stretch of road expressed in meters per second
  - `crossTime` It is the estimated crossing time of the edge for each transportation mode. If the cross time is ∞ then it means that the road has an access restriction in that direction, such has no entry signs.

    It was used to use with the *GDS*' implementation of $A*$, since we have implemented our version of $A*$ we only check for ∞
  - `class` The road class in the hierarchy where *motorway* is at the top.

For the disruption handling we have the following nodes and relationships:

- **Disruption** a node in the graph containing all the informations about an active disruption

```
(:Disruption {
  severity: "Moderate",
  subCategory: "TfL",
  centrum: point({srid:4326, x:-0.27671, y:51.489055}),
  closed: FALSE,
  update: "Delays are possible.",
  comment: "....",
  updateTime: 2023-01-25T07:40:25,
  id: "TIMS-265653",
  category: "Works",
  radius: 123.0,
  ttl: 575979
})
```

  We save in this edge all the information that could be of immediate help to the user, to avoid making another query to the other databases when requesting a preview of the data. In particular we store:

  - `ttl` It is the *Time to live*, that is the number of seconds that remains to programmed end of the disruption, so that if there are problems in the *driver TIMS* the *DBMS* can drop it

- **isDisrupted** is a relation between a Point and a Disruption, telling that the road is being affected by a disruption.

  We don't store any information on those edges.

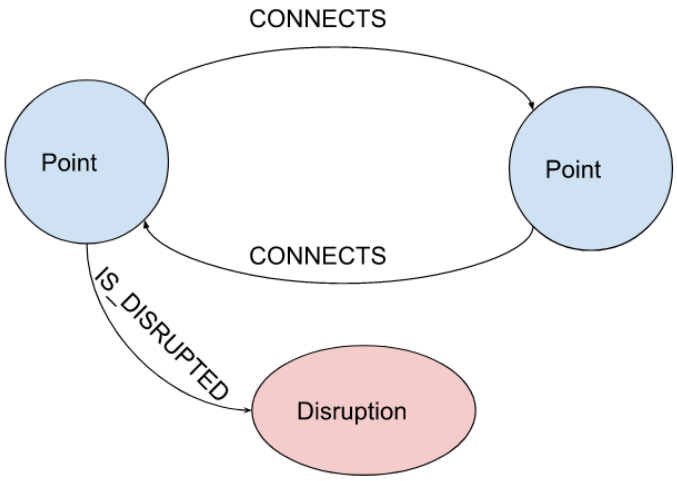
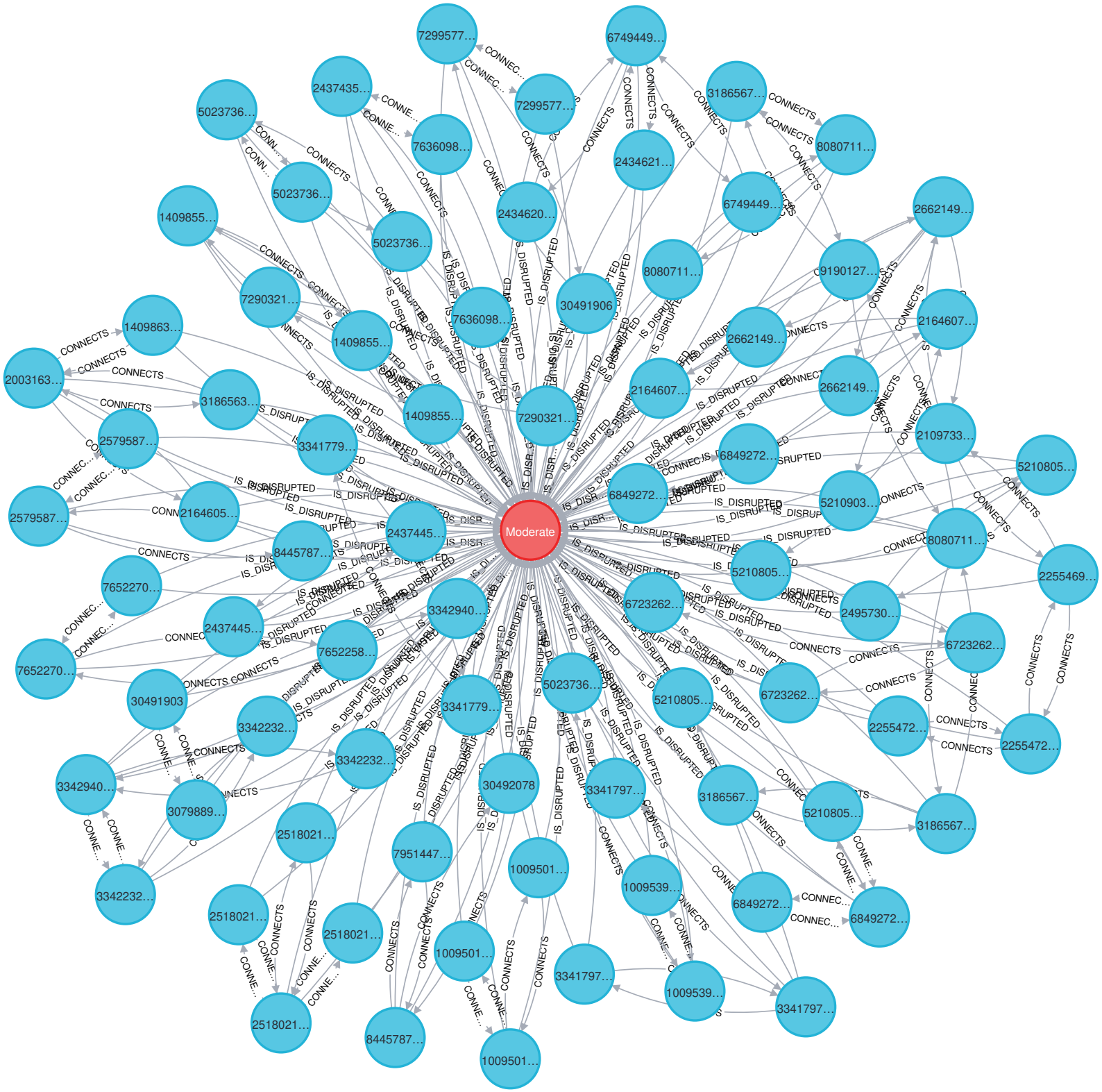


Figure 8.1: A possible portion of graph

Example Now it follows a snapshot of the graph:


Moderate
7299577...
6749449...
2437435...
3186567...
5023736...
7636098...
8080711...
2434621...
2434620...
1409855...
2662149...
9190127...
7290321...
30491906
1409863...
2164607...
2003163...
3186563...
2579587...
3341779...
2109733...
5210805...
2164605...
6849272...
5210903...
8445787...
2437445...
7652270...
3342940...
6723262...
2495730...
7652258...
5023736...
30491903
3342232...
3341779...
2255472...
3079889...
30492078
3341797...
2518021...
7951447...
1009501...
1009539...
8445787...

Chapter 9

Distributed database design

The distributed database's design is suggested by requirements of the applications.

9.1 Replica set

MongoDB In order to ensure that all servers have the same data and to improve performance by allowing multiple servers to process queries, and can also provide a level of fault tolerance, as data can be retrieved from a replica server if the primary server goes down we decided to have three virual replicas with on each one a MongoDB instance hosted.

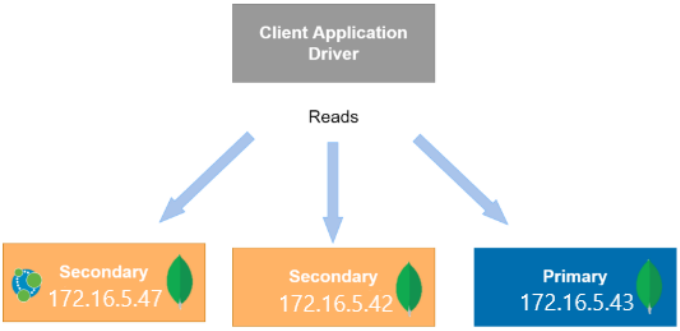


Figure 9.1

Neo4J Regarding the Neo4J part, it is present only on one replica.

Composition The three virtual machines are kindly provided by the University of Pisa and the replica set is composed of a primary replica that acts as the server that takes client requests, and two secondaries which are the servers that keep copies of the primary's data.

Virtual Machine	IP address	Port	OS
Replica-0	172.16.5.43	27017	Ubuntu
Replica-1	172.16.5.47	27017	Ubuntu
Replica-2	172.16.5.42	27017	Ubuntu

Table 9.1: Virtual machines settings

9.2 Replica Configuration

The configuration is shown below:

```
rsconf = {
  _id: "londonSafeTravelSet",
  members: [
  {
    _id: 0,
    host: "172.16.5.43:27017",
    priority:1
  },
  {
    _id: 1,
    host: "172.16.5.47:27017",
      priority:2
    },
  {
    _id: 2,
     host: "172.16.5.42:27017",
     priority:5
  }]
};

rs.initiate(rsconf);
```

After careful consideration, it was determined that the virtual machine with the IP address 172.16.4.43 should be given the highest priority and will serve as the primary replica, unless any unforeseen issues arise. As previously noted in regards to the handling of the CAP theorem, the application in question has a high ratio of read to write operations. Thus, in order to guarantee high availability and protection against partitioning, it was decided to adopt the Eventual Consistency paradigm. It should be noted, however, that in the event of partitioning, there may be instances where data returned may not be the most recent version.

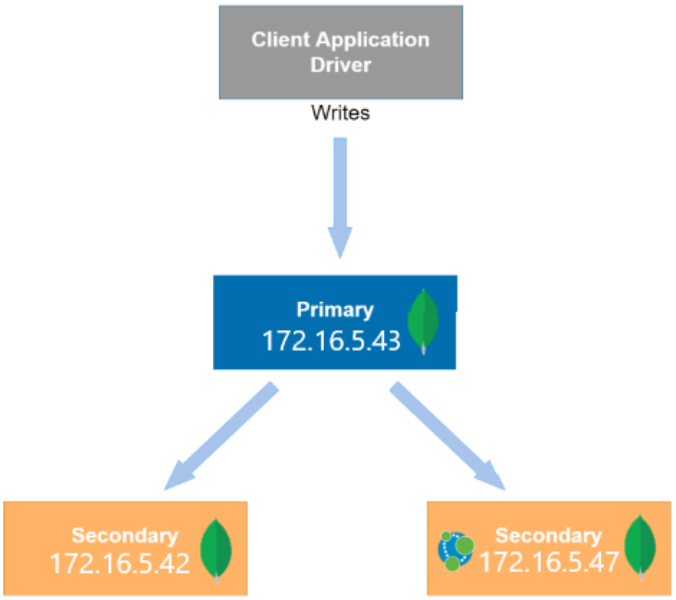


Figure 9.2

9.3 Replica crash

Crash In the event of failure of the primary node, the responsibility of primary role will be transferred to one of the two secondary replicas. As priorities have been assigned to each replica, it has been predetermined which of the secondary replicas will assume the role of primary. In this specific implementation, should the primary node become unavailable, the virtual machine with the IP address 172.16.4.47 will be designated as the new primary node.

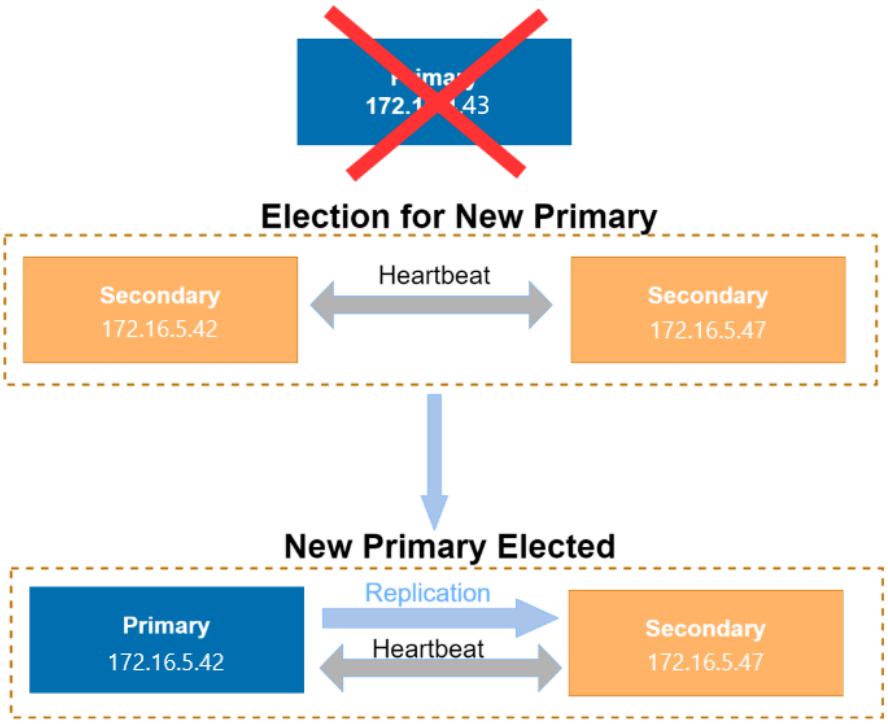


Figure 9.3

Chapter 10

Overall platform architecture

The application has been developed utilizing the Java programming language and the Intellij integrated development environment. As outlined in the Data Model section, MongoDB and Neo4j were chosen as the NoSQL database management systems for data storage and management. It was previously stated that there are three replicas implemented for MongoDB, and a single replica for Neo4j.

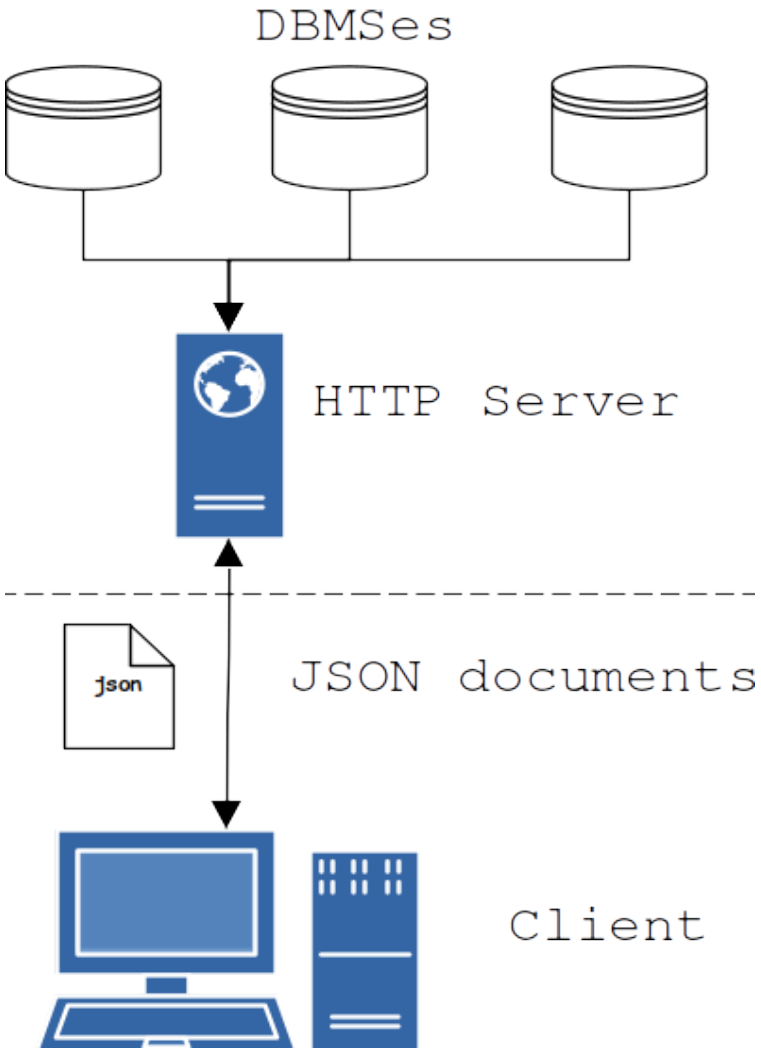


Figure 10.1: Application architecture

Chapter 11

Technologies used

The application was created utilizing the capabilities of various powerful technologies such as

- **Java Swing** for creating an interactive and user-friendly interface and a couple of libraries:
  - `jxMapViewer2`
  - `jxFreeChart`
- the **MongoDB driver** for effectively managing and manipulating data stored in MongoDB
- the **Neo4j driver** for managing and querying graph data
- **Gson** for handling JSON data in a convenient and efficient manner

All of these technologies were integrated seamlessly to deliver a robust and high-performing application.

REST API Moreover, we make use of the powerful Java standard library to implement a RESTful API using the built-in web server. The Java standard library provides a simple yet effective way to create and manage a web server, allowing for the creation of a lightweight and efficient REST API that adheres to industry standards. The use of the built-in web server eliminates the need for additional third-party dependencies, making the development process more streamlined and the deployment process more manageable.

This choice of technology also ensures compatibility and ease of integration with other Java based systems. Additionally, Gson library is used for serialization and deserialization of Java data access objects (DAOs) to and from JSON, providing a convenient and efficient way to handle data transfer between the application and the API.

(a) (b) (c)

Figure 11.1: Logo

Part IV

Implementation

Chapter 12

Java application

Introduction All parts of the applications are written in *Java* using *Maven* as a building system to handle all the dependencies used in the project.

12.1 Source tree

Version Control We used *git* as the version control software to develop our system. The reader can find the public repository on *GitHub* at following URL: `https://github.com/scarburato/LargeScaleDBsProject`

All the code is stored into the *londonSafeTravel* package
This is the source tree of the Java packages:

```
src/libCommon/src/main/java/londonSafeTravel/
+-- client
| +-- gui
| | +-- AdministrationDialog.form
| | +-- AdministrationDialog.java
| | +-- AnalyticsMap.form
| | +-- AnalyticsMap.java
| | +-- DisruptionDialog.form
| | +-- DisruptionDialog.java
| | +-- DisruptionWaypoint.java
| | +-- GlobalPainter.java
| | +-- GraphLine.java
| | +-- HeatmapPainter.java
| | +-- MainApp.form
| | +-- MainApp.java
| | +-- POIDialog.form
| | +-- POIDialog.java
| | +-- POIEventHandler.java
| | +-- POIRenderer.java
| | +-- POIWaypoint.java
| +-- net
| +-- DisruptionsRequest.java
| +-- HeatmapRequest.java
| +-- LineGraphRequest.java
| +-- POIRequest.java
| +-- QueryPointRequest.java
| +-- RoutingRequest.java
| +-- SearchRequest.java
| +-- StatTableRequest.java
+-- dbms
| +-- bsonCodecs
| | +-- POICodec.java
```

```
| +-- document
| | +-- ConnectionMongoDB.java
| | +-- DisruptionStatsDAO.java
| | +-- LineGraphDAO.java
| | +-- DisruptionDAO.java
| | +-- PointOfInterestDAO.java
| | +-- TransitStopDAO.java
| +-- graph
| +-- DisruptionDAO.java
| +-- PointDAO.java
| +-- RoutingDAO.java
| +-- WayDAO.java
+-- driver
| +-- tims
| +-- RoadDisruptionUpdate.java
+-- gsonUtils
| +-- gsonCodecs
| | +-- MongoGeometryCodec.java
| | +-- POICodec.java
| | +-- POIOSMCodec.java
| +-- GsonFactory.java
+-- OSMImporter
| +-- POI
| | +-- POIFactory.java
| | +-- POI.java
| | +-- Point.java
| | +-- Way.java
| +-- StreamImporter.java
| +-- WaysFactory.java
+-- schema
| +-- document
| | +-- Disruption.java
| | +-- HeatmapComputation.java
| | +-- LineGraphEntry.java
| | +-- poi
| | +-- PointOfInterest.java
| | +-- PointOfInterestOSM.java
| +-- GeoFactory.java
| +-- graph
| | +-- Disruption.java
| | +-- Point.java
| | +-- RoutingHop.java
| | +-- Way.java
| +-- Location.java
+-- server
   +-- ExceptionTester.java
   +-- Handler.java
   +-- HeatmapHandler.java
   +-- LineGraphHandler.java
   +-- POIHandler.java
   +-- QueryDisruptionHandler.java
   +-- QueryPointHandler.java
   +-- QueryStatTableHandler.java
   +-- RoutingHandler.java
   +-- SearchHandler.java
   +-- Server.java
```

.

12.1.1 Packages

The most relevant packages:

- **schema** Contains the **data transfer objects** used to get or send data to the databases
- **dbms** Contains the **data access objects** used to access the databases
- **driver.tims** Contains the client and all its relevant classes for the **TIMS** user
- **gsonUtilis** Contains the serialization and de-serialization facilities for *Gson*, the library used to exchange *JSON*s between client and server
- **OSMImporter** Contains a support program used to import the relevant *OpenStreetMap*'s data into the various databases.
- **server** Contains the server application, used to provide remote access to the database to the *user* and the *statistician*
- **client** Contains the client application, both the for the *user* and the *statistician*. It has two sub-packages:
  - **net** handles all the network requests to the server
  - **gui** contains all the stuff related to the user interface, that is implemented in *Java Swing*

12.1.2 Java model classes

Design All model classes, which are stored into the *schema* package, are *POJO*s classes that are ordinary Java objects, not bound by any special restriction.

MongoDB data access objects

Disruption This object represents the documents stored into the *Disruption* collection of *MongoDB*

```
1  public class Disruption {
2
3     public static class Update{
4        public Date start;
5        public Date end;
6        public String message;
7     }
8     public static class Street{
9        public String name;
10       public Boolean closure;
11       public String direction;
12       public List<LineString> segments;
13
14       public Street() {segments = new ArrayList<>();}
15    }
16    @BsonProperty
17    public String id;
18    @BsonProperty
19    public String type;
20    @BsonProperty
21    public Date start;
22    @BsonProperty
23    public Date end;
24    @BsonProperty
25    public Point coordinates;
26    @BsonProperty
27    public Geometry boundaries;
```

```
28
29      @BsonProperty
30      public String category;
31
32      @BsonProperty
33      public String subCategory;
34      @BsonProperty
35      public String severity;
36      @BsonProperty
37      public List<Update> updates;
38
39      @BsonProperty
40      public List<Street> streets;
41      @BsonProperty("closure")
42      public Boolean closure; // se stazione (0,1), se in street esiste una
            ↪ closure(0,1)
43
44      public Disruption()
45      {
46          updates = new ArrayList<>();
47          streets = new ArrayList<>();
48      }
49  }
```

Point of Interest Those objects represents the documents stored into the *PointOfInterest* collection. Originally it was planned to have two kinds of *POI*s stored into the database: the ones from *OpenStreetMap* and another kind, called *TransitStop*, that was supposed to store specific information about public transportation's stop.

Currently only the first is present in the application, while the latter was cut; so in the codes still survives the generalization of those two classes:

PointOfInterest The super-class

```
1  public class PointOfInterest {
2      public String getType() {
3          return "GENERIC";
4      };
5      @BsonProperty
6      public String poiID;
7      @BsonProperty
8      public String name;
9      @BsonProperty
10     public Point coordinates;
11 }
```

PointOfInterestOSM The only kind of document currently used in the application

```
1  public class PointOfInterestOSM extends PointOfInterest{
2      public HashMap<String, String> tags;
3      public Geometry perimeter;
4
5      public String getType() {
6          return "OSM-POI";
7      }
8  }
```

MongoDB data access objects for queries

Heatmap The *HeatmapComputation* class that contains the result of the heatmap query computation, the (latitude, longitude) could be thought as the indexes of a matrix.

```
public class HeatmapComputation {
   @BsonProperty
   public double latitude;
   @BsonProperty
   public double longitude;
   @BsonProperty
   public long count;
}
```

Time series graph The *LineGraphEntry* class contains the points of the time series computed in the query

```
public class LineGraphEntry {
   @BsonProperty("_id")
   public long hour;

   @BsonProperty
   public double count;
}
```

Neo4j data access objects

Active Disruption An active disruption is represented by the class *Disruption*

```
public class Disruption {
   public Location centrum;
   public String id;
   public Double radius;
   public String severity;
   public long ttl;

   public String category;
   public String subCategory;

   public String comment;

   public String update;
   public Date updateTime;
   public boolean closed;
}
```

Point A node with label *Point*, also known as an *Intersection*, on the graph is represented by the *POJO* class *Point*

```
public class Point {
   @Override
   public String toString() {
      return "Point{" +
            "id=" + id +
            ",␣location=" + location +
            '}';
   }

   private final long id;
   public Location location;

   public Point(long id, double latitude, double longitude){
      this.id=id;
      this.location=new Location(latitude,longitude);
   }
```

```
17
18      public long getId() {
19          return id;
20      }
21
22      public Location getLocation() {
23          return location;
24      }
25  }
```

Way A relationship between two *Points* with label *CONNECTS* is represented by the *POJO* class *Way*

```
1  public class Way {
2      public long id;
3      public Point p1;
4      public Point p2;
5      public String name;
6      public double maxSpeed;
7      public String roadClass;
8      public HashMap<String, Double> crossTimes;
9  }
```

Neo4j data access objects for queries

Routing A routing hop in the path computed by the routing procedure is represented by the class *RoutingHop*

```
1  public class RoutingHop {
2      public Point point;
3      public double time;
4
5      public RoutingHop(Point point, double time) {
6          this.point = point;
7          this.time = time;
8      }
9  }
```

12.2 Considerations on geographical data

Built-in types Both MongoDB and Neo4j's drivers use their own Java classes to represents geographical data, which are not compatible directly with each other. There's also Filosgnagna's GeoJson classes used by the *driver TIMS* user to parse the *JSON*s data provided by *Transport for London*.

Solution We introduced a fourth class called *Location* that is used in the *POJO*s where appropriate, in place of the driver's native types.

```
1  public class Location {
2      public static class Polygon {
3          public Polygon(List<Location> outer, List<Location> inner) {
4              this.outer = outer;
5              this.inner = inner;
6          }
7
8          public Polygon() {};
9
10         public List<Location> outer;
11         public List<Location> inner;
```

```
12      }
13
14      private double longitude;
15      private double latitude;
16
17      // create and initialize a point with given name and
18      // (latitude, longitude) specified in degrees
19      public Location(double latitude, double longitude) {
20         this.latitude = latitude;
21         this.longitude = longitude;
22      }
23
24      public double metricNorm(Location l2) {
25         double earthRadiusKm = 6371.0 * 1000;
26
27         double lat1Rad = Math.toRadians(latitude);
28         double lon1Rad = Math.toRadians(longitude);
29         double lat2Rad = Math.toRadians(l2.latitude);
30         double lon2Rad = Math.toRadians(l2.longitude);
31
32         return earthRadiusKm * Math.acos(Math.sin(lat1Rad) * Math.sin(lat2Rad) +
33               Math.cos(lat1Rad) * Math.cos(lat2Rad) *
34                     Math.cos(lon1Rad - lon2Rad));
35      }
36
37      // return distance between this location and that location
38      // measured in statute miles
39      public double distanceTo(Location that) {
40         double STATUTE_MILES_PER_NAUTICAL_MILE = 1.15077945;
41         double lat1 = Math.toRadians(this.latitude);
42         double lon1 = Math.toRadians(this.longitude);
43         double lat2 = Math.toRadians(that.latitude);
44         double lon2 = Math.toRadians(that.longitude);
45
46         // great circle distance in radians, using law of cosines formula
47         double angle = Math.acos(Math.sin(lat1) * Math.sin(lat2)
48               + Math.cos(lat1) * Math.cos(lat2) * Math.cos(lon1 - lon2));
49
50         // each degree on a great circle of Earth is 60 nautical miles
51         double nauticalMiles = 60 * Math.toDegrees(angle);
52         double statuteMiles = STATUTE_MILES_PER_NAUTICAL_MILE * nauticalMiles;
53         return statuteMiles;
54      }
55
56      // return string representation of this point
57      public String toString() {
58         return "␣(" + latitude + ",␣" + longitude + ")";
59      }
60
61      public double getLongitude() {
62         return longitude;
63      }
64
65      public double getLatitude() {
66         return latitude;
67      }
68
69      public void setLongitude(double longitude) {
70         this.longitude = longitude;
71      }
72
```

```
73      public void setLatitude(double latitude) {
74          this.latitude = latitude;
75      }
76
77      // test client
78      public static void main(String[] args) {
79          Location loc1 = new Location(40.366633, 74.640832);
80          Location loc2 = new Location(42.443087, 76.488707);
81          double distance = loc1.distanceTo(loc2);
82          System.out.printf("%6.3f␣miles␣from\n", distance);
83          System.out.println("test" + loc1.metricNorm(loc2));
84          System.out.println(loc1 + "␣to␣" + loc2);
85      }
86  }
```

Interoperability To ease conversions between those four representation a *GeoFactory* class was added, it contains utility methods to convert from a format to another the most common geometries used in the application.

```
1   public class GeoFactory {
2       public static Location fromNeo4j(Point p) {
3           return new Location(p.y(), p.x());
4       }
5
6       public static Location fromFilosgangaToLocation(
7               com.github.filosganga.geogson.model.Point p) {
8           return new Location(p.lat(), p.lon());
9       }
10
11      public static Location fromMongo(com.mongodb.client.model.geojson.Point
            ↪ point) {
12          return new Location(
13                  point.getCoordinates().getValues().get(1),
14                  point.getCoordinates().getValues().get(0)
15          );
16      }
17
18      public static com.mongodb.client.model.geojson.Point convertToMongo(Location
            ↪ location) {
19          return new com.mongodb.client.model.geojson.Point(new Position(
20                  location.getLongitude(),
21                  location.getLatitude()
22          ));
23      }
24
25      public static LineString convertToMongo(List<Location> line) {
26          return new LineString(
27                  line.stream()
28                          .map(GeoFactory::convertToMongo)
29                          .map(com.mongodb.client.model.geojson.Point::getPosition)
30                          .collect(Collectors.toList())
31          );
32      }
33
34      public static Polygon convertToMongo(Location.Polygon poly) {
35          return new Polygon(convertToMongo2(poly));
36      }
37
38      public static PolygonCoordinates convertToMongo2(Location.Polygon poly) {
39          return new PolygonCoordinates(
40                  poly.outer.stream()
```

```
41                  .map(GeoFactory::convertToMongo)
42                  .map(com.mongodb.client.model.geojson.Point::getPosition)
43                  .collect(Collectors.toList()),
44            poly.inner.stream()
45                  .map(GeoFactory::convertToMongo)
46                  .map(com.mongodb.client.model.geojson.Point::getPosition)
47                  .collect(Collectors.toList())
48      );
49   }
50
51   public static MultiPolygon convertToMongo2(List<Location.Polygon> polygons)
         ↪ {
52      return new MultiPolygon(
53            polygons.stream()
54                  .map(GeoFactory::convertToMongo2)
55                  .collect(Collectors.toList())
56      );
57   }
58
59   public static com.mongodb.client.model.geojson.Point fromFilosgangaToMongo(
60         com.github.filosganga.geogson.model.Point point) {
61      return convertToMongo(new Location(point.lat(), point.lon()));
62   }
63
64   public static LineString fromFilosgangaToMongo(
65         com.github.filosganga.geogson.model.LineString line) {
66      return new LineString(
67            StreamSupport.stream(line.points().spliterator(), false)
68                  .map(GeoFactory::fromFilosgangaToMongo)
69                  .map(com.mongodb.client.model.geojson.Point::getPosition)
70                  .collect(Collectors.toList()));
71   }
72
73   public static Polygon fromFilosgangaToMongo(
74         com.github.filosganga.geogson.model.Polygon polygon) {
75      if (polygon.holes().iterator().hasNext())
76         return new Polygon(
77               fromFilosgangaToMongo(
78                     polygon.linearRings().iterator().next())
79                     .getCoordinates(),
80
81               fromFilosgangaToMongo(
82                     polygon.holes().iterator().next())
83                     .getCoordinates()
84         );
85      else
86         return new Polygon(
87               fromFilosgangaToMongo(
88                     polygon.linearRings().iterator().next()
89               ).getCoordinates()
90         );
91   }
92
93   public static MultiPolygon fromFilosgangaToMongo(
94         com.github.filosganga.geogson.model.MultiPolygon multiPolygon) {
95      return new MultiPolygon(
96            StreamSupport.stream(multiPolygon.polygons().spliterator(), false)
97                  .map(GeoFactory::fromFilosgangaToMongo)
98                  .map(Polygon::getCoordinates)
99                  .collect(Collectors.toList())
100     );
```

```
101     }
102
103     public static com.github.filosganga.geogson.model.Point toSgagna(Location p)
            ↪ {
104         return com.github.filosganga.geogson.model.Point
105                 .from(p.getLongitude(), p.getLatitude());
106     }
107 }
```

12.3 Maven build system

The configuration file used to build the application:

```
1   <packaging>jar</packaging>
2
3   <name>libCommon</name>
4   <url>http://maven.apache.org</url>
5
6   <properties>
7     <maven.compiler.source>19</maven.compiler.source>
8     <maven.compiler.target>19</maven.compiler.target>
9     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
10  </properties>
11
12  <build>
13    <plugins>
14      <!-- any other plugins -->
15      <plugin>
16        <artifactId>maven-assembly-plugin</artifactId>
17        <executions>
18          <execution>
19            <phase>package</phase>
20            <goals>
21              <goal>single</goal>
22            </goals>
23          </execution>
24        </executions>
25        <configuration>
26          <descriptorRefs>
27            <descriptorRef>jar-with-dependencies</descriptorRef>
28          </descriptorRefs>
29        </configuration>
30      </plugin>
31    </plugins>
32  </build>
33
34
35  <dependencies>
36    <dependency>
37      <groupId>org.apache.httpcomponents.client5</groupId>
38      <artifactId>httpclient5</artifactId>
39      <version>5.2.1</version>
40    </dependency>
41    <dependency>
42      <groupId>jfree</groupId>
43      <artifactId>jfreechart</artifactId>
44      <version>1.0.13</version>
45    </dependency>
46    <dependency>
47      <groupId>org.mongodb</groupId>
```

```
48      <artifactId>mongodb-driver-sync</artifactId>
49      <version>RELEASE</version>
50    </dependency>
51    <dependency>
52      <groupId>org.neo4j.driver</groupId>
53      <artifactId>neo4j-java-driver</artifactId>
54      <version>5.3.1</version>
55    </dependency>
56    <dependency>
57      <groupId>com.google.code.gson</groupId>
58      <artifactId>gson</artifactId>
59      <version>2.8.9</version>
60    </dependency>
61    <!-- FOR GEOJSON -->
62    <dependency>
63      <groupId>com.github.filosganga</groupId>
64      <artifactId>geogson-core</artifactId>
65      <version>1.2.21</version>
66    </dependency>
67    <!-- FOR MAP -->
68    <dependency>
69      <groupId>org.jxmapviewer</groupId>
70      <artifactId>jxmapviewer2</artifactId>
71      <version>2.5</version>
72    </dependency>
73  </dependencies>
74
75 </project>
```

Chapter 13

DocumentDB CRUD operations

Here below are reported the principal CRUD operations.

13.1 Disruption: Create and Update

The insert operation in the DocumentDB is performed by the TIMS API. At regular intervals, every 10 minutes, the API is called and returns a list of disruption updates. The Set operation inserts a new disruption if the returned ID is not already present in the DB or updates it otherwise. Here below there is the Java code:

```
1   public void set(Disruption d){
2      var docs = collection.find(eq("id", d.id));
3      var resultDoc = docs.first();
4
5      try {
6         if (resultDoc == null) {
7            collection.insertOne(d);
8         }else{
9            collection.replaceOne(eq("id", d.id), d);
10        }
11     }catch (MongoException me) {
12        System.err.println("Unable to update due to an error: " + me);
13     }
14  }
```

13.2 Disruption: Read

The Get method must return a disruption when invoked, given its ID as input.

```
1   public Disruption get(String id) {
2      return collection.find(eq("id", id)).first();
3   }
```

13.3 Point of interest: Create

To make the service usable, it was necessary to enter the POIs in the document database. A POI is entered using the following function:

```
1   public void insert(PointOfInterest poi) {
2      collection.insertOne(poi);
3   }
```

13.4 Point of Interest: Read

We have two types of POI searches; regarding the former, it is useful to have a function that returns the data of a given POI given its ID. For the operation of the application it is also useful to have a function that given as parameters:

- Maximum latitude
- Minimum latitude
- Maximum longitude
- Minimum longitude

returns the list of POIs in the area described by the previous values. The functions are the following:

```
1    public PointOfInterest getPOI(String id) {
2       Bson match = eq("poiID", id);
3       return collection.find(match).first();
4    }
5
6    public List<PointOfInterest> selectPOIsInArea(double minLong,
7                                      double maxLong, double minLat,
8                                      double maxLat)
9    {
10      assert (minLong < maxLong);
11      assert (minLat < maxLat);
12
13      Polygon region = new Polygon(Arrays.asList(
14            new Position(minLong, minLat),
15            new Position(maxLong, minLat),
16            new Position(maxLong, maxLat),
17            new Position(minLong, maxLat),
18            new Position(minLong, minLat)
19      ));
20
21      Bson myMatch = geoWithin("coordinates", region);
22      ArrayList<PointOfInterest> results = new ArrayList<>();
23
24      collection.find(myMatch).forEach(results::add);
25      return results;
```

Chapter 14

Graph Database CRUD operations

14.1 Bootstrap

Since the transaction manager of Neo4j is not able to cope with huge amount of data, it was necessary to insert the data into the database when the DBMS was halted, in a sort of cold insert, using the bundled utility *neo4j-admin* with the following shell script:

```
1  #!/bin/sh
2
3  bin/neo4j-admin database import full \
4    --overwrite-destination \
5    --nodes import/nodes.csv \
6    --relationships import/ways.csv \
7    --id-type=integer \
8    --high-parallel-io=on \
9    --max-off-heap-memory=16G \
10   --threads=16
```

14.2 Point: Create

To insert a Point in the graph database the following code is used:

```
1     public void addNode(final Point p) {
2        try (Session session = driver.session()) {
3           session.executeWriteWithoutResult(tx -> createPlaceNode(tx, p.getId(),
                ↪ p.location.getLatitude(), p.location.getLongitude()));
4        }
5     }
6
7     private static final Query PLACE_NODE = new Query("CREATE␣(p:Point{id:$id,␣
          ↪ lat:$lat,␣longitude:$longitude})");
8
9     private void createPlaceNode(TransactionContext tx, long id, double lat,
          ↪ double longitude) {
10       Query q = PLACE_NODE.withParameters(parameters("id", id, "lat", lat, "
             ↪ longitude", longitude));
11       tx.run(q);
12    }
```

14.3 Way: Create

To insert a Way in the graph database the following code is used:

```
1     public void addWays(Collection<Way> ways) {
2        try (var session = driver.session()) {
3           session.executeWriteWithoutResult(tx -> createWays(tx, ways));
4        }
5     }
6
7     private void createWays(TransactionContext tx, Collection<Way> ways) {
8        ways.forEach(way -> {
9           tx.run(
10                "MERGE␣(p1:␣Point␣{id:␣$id1})" +
11                      "MERGE␣(p2:␣Point␣{id:␣$id2})" +
12                      "MERGE␣(p1)-[:CONNECTS␣{" +
13                      "␣␣␣name:␣$name,␣class:␣$class,␣maxspeed:␣$speed," +
14                      "␣␣␣crossTimeFoot:␣$crossFoot,␣crossTimeBicycle:␣
                            ↪ $crossBicycle,␣crossTimeMotorVehicle:␣
                            ↪ $crossMotorVehicle" +
15                      "}]->(p2)" +
16                      "MERGE␣(p2)-[:CONNECTS␣{" +
17                      "␣␣␣name:␣$name,␣class:␣$class,␣maxspeed:␣$speed," +
18                      "␣␣␣crossTimeFoot:␣$crossFoot,␣crossTimeBicycle:␣
                            ↪ $crossBicycle,␣crossTimeMotorVehicle:␣
                            ↪ $crossMotorVehicle" +
19                      "}]->(p1)␣" +
20                      "ON␣CREATE␣SET␣p1.coord␣=␣point({longitude:␣$lon1,␣
                            ↪ latitude:␣$lat1})␣" +
21                      "ON␣CREATE␣SET␣p1.lat␣=␣$lat1␣" +
22                      "ON␣CREATE␣SET␣p1.lon␣=␣$lon1␣" +
23                      "ON␣CREATE␣SET␣p2.coord␣=␣point({longitude:␣$lon2,␣
                            ↪ latitude:␣$lat2})" +
24                      "ON␣CREATE␣SET␣p2.lat␣=␣$lat2,␣p2.lon␣=␣$lon2",
25                parameters(
26                      "id1", way.p1.getId(),
27                      "lat1", way.p1.getLocation().getLatitude(),
28                      "lon1", way.p1.getLocation().getLongitude(),
29                      "id2", way.p2.getId(),
30                      "lat2", way.p2.getLocation().getLatitude(),
31                      "lon2", way.p2.getLocation().getLongitude(),
32                      //"wid", way.id,
33                      "name", way.name,
34                      "crossFoot", way.crossTimes.get("foot"),
35                      "crossBicycle", way.crossTimes.get("bicycle"),
36                      "crossMotorVehicle", way.crossTimes.get("motor_vehicle"),
37                      "speed", way.maxSpeed,
38                      "class", way.roadClass
39                )
40          );
41       });
42    }
```

14.4 Disruption: Create and Update

The code below is used both for create and update a disruption. If the relative ID is already present in the DB, we simply update the data, otherwise we insert a new disruption.

```
1        try (Session session = driver.session()) {
2           session.executeWriteWithoutResult(writeDisruption(disruption));
3        }
4     }
5
6     public void createClosures(List<Disruption> disruptions) {
```

```
7         try (Session session = driver.session()) {
8            disruptions.forEach(disruption -> session.executeWriteWithoutResult(
                 ↪ writeDisruption(disruption)));
9         }
10     }
11
12     private static Consumer<TransactionContext> writeDisruption(Disruption
           ↪ disruption) {
13        return transactionContext -> {
14           Query q = CREATE_CLOSURE.withParameters(parameters(
15                 "id", disruption.id,
16                 "lat", disruption.centrum.getLatitude(),
17                 "lon", disruption.centrum.getLongitude(),
18                 "radius", disruption.radius,
19                 "ttl", disruption.ttl,
20                 "severity", disruption.severity,
21                 "category", disruption.category,
22                 "subcategory", disruption.subCategory,
23                 "comment", disruption.comment,
24                 "closed", disruption.closed,
25                 "update", disruption.update,
26                 "updateTime", disruption.updateTime.toInstant()
27                       .atZone(ZoneId.systemDefault())
28                       .toLocalDateTime()
29           ));
30           transactionContext.run(q);
31        };
32     }
```

14.5 Disruption: Read

To find informations about active disruptions, the following code is used:

```
1         ArrayList<Disruption> disruptions= new ArrayList<>();
2         try (Session session = driver.session()) {
3            var res= session.run(FIND_ACTIVE_DISRUPTIONS);
4
5            res.forEachRemaining(record -> {
6               var d = new Disruption();
7               d.id = record.get("d").get("id").asString();
8               d.centrum = GeoFactory.fromNeo4j(record.get("d").get("centrum").
                    ↪ asPoint());
9               d.radius = record.get("d").get("radius").asDouble();
10              d.severity = record.get("d").get("severity").asString();
11              d.ttl = record.get("d").get("ttl").asLong();
12              d.severity = record.get("d").get("severity").asString();
13              d.category = record.get("d").get("category").asString();
14              d.subCategory = record.get("d").get("severity").asString();
15              d.comment = record.get("d").get("comment").asString();
16              d.update = record.get("d").get("update").asString();
17              d.updateTime = java.sql.Timestamp.valueOf(record.get("d").get("
                    ↪ updateTime").asLocalDateTime());
18              d.closed = record.get("d").get("closed").asBoolean(false);
19              disruptions.add(d);
20           });
21        }
22        return disruptions;
23     }
24  }
```

14.6 Disruption: Delete

When a disruption is no longer active, we need to remove it from graph database. To perform this, the following code is used:

```
1           "MATCH␣(d:Disruption␣{id:␣$id})␣" +
2                 "DETACH␣DELETE␣d"
3     );
4
5     public void deleteClosure(String id) {
6        try (Session session = driver.session()) {
7           session.executeWriteWithoutResult(transactionContext -> {
8              Query q = DELETE_CLOSURE.withParameters(parameters(
9                    "id", id
10             ));
11             transactionContext.run(q);
12          });
13       }
14    }
```

Chapter 15

Queries on graph database

In this chapter are presented the relevant queries for the Neo4j graph database.

15.1 Routing

Our routing query is used to compute a suboptimal path between two points on the map, given as input to the query, using the algorithm known as **Anytime A***.

```
MATCH (s:Point{id: $start})
MATCH (e:Point{id: $end})
CALL londonSafeTravel.route.anytime(s, e, $type, $maxspeed, 12.5)
YIELD index, node, time
RETURN index, node AS waypoint, time
ORDER BY index DESCENDING
```

Figure 15.1: Cypher query

The procedure *lodonSafeTravel.route.anytime* has been implemented as show below:

```
    }
    @Procedure(value = "londonSafeTravel.route.anytime", mode = Mode.READ)
    @Description("Finds␣the␣sub-optimal␣path␣between␣two␣POINTS")
    public Stream<HopRecord> route2(
          @Name("start") Node start,
          @Name("end") Node end,
          @Name("crossTimeField") String crossTimeField,
          @Name("considerDisruptions") boolean considerDisruptions,
          @Name("maxSpeed") double maxspeed,
          @Name("w") double weight
    ) {
       if(!start.hasLabel(POINT))
          throw new IllegalArgumentException("'start'␣does␣not␣have␣'Point'␣as␣a
              ↪ ␣label");

       if(!end.hasLabel(POINT))
          throw new IllegalArgumentException("'end'␣does␣not␣have␣'Point'␣as␣a␣
              ↪ label");

       if(weight < 1.0 || weight == Double.POSITIVE_INFINITY)
          throw new IllegalArgumentException("'weight'␣must␣be␣in␣[1,␣+Infinity)
              ↪ ");

       TreeSet<RouteNode> openSetHeap = new TreeSet<>();
       HashMap<Long, RouteNode> openSet = new HashMap<>(0xfff, 0.666f);
```

```
23        HashMap<Long, RouteNode> closedSet = new HashMap<>(0xfeee, 0.666f);
24
25        var startNode = new RouteNode(
26              new Cost(0, weight * heuristic(start, end, maxspeed)), start);
27        openSetHeap.add(startNode);
28        openSet.put(startNode.getId(), startNode);
29
30        log.info("Starting␣route␣from␣" + getIdOf(start) + "\tto␣" + getIdOf(end)
              ↪ );
31
32        RouteNode current = null;
33        while(! openSet.isEmpty()) {
34           assert (openSet.size() == openSetHeap.size());
35
36           // Get current node
37           current = openSetHeap.pollFirst(); // O(log n)
38           openSet.remove(current.getId()); // ~O(1)
39
40           // Move from open to closed
41           closedSet.put(current.getId(), current); // ~O(1)
42
43           // Found the solution HALT!
44           if(current.getId() == getIdOf(end))
45              break;
46
47           var hops = current.node.getRelationships(Direction.OUTGOING, CONNECTS)
                 ↪ ; // O(1), I hope?
48
49           for(var way : hops) {
50              // NO ENTRY IN HERE!
51              final double crossTimeStored = (Double) way.getProperty(
                    ↪ crossTimeField);
52              if(crossTimeStored == Double.POSITIVE_INFINITY)
53                 continue;
54
55              // Successor info
56              Node successor = way.getEndNode();
57              final long successorId = getIdOf(successor);
58
59              // cost for this edge
60              double crossTime = crossTime(way, current.node, successor, maxspeed
                    ↪ );
61
62              // Look for disruption
63              if(considerDisruptions)
64                 crossTime = considerDisruptions(successor, crossTime);
65
66              // Adjust for minor roads
67              if(crossTimeField.equals("crossTimeMotorVehicle")) {
68                 Double factor = timeWeights.get((String)way.getProperty("class")
                       ↪ );
69                 if(factor != null)
70                    crossTime *= factor;
71              }
72
73              // If big intersection, add 5 seconds
74              if(successor.getDegree(CONNECTS, Direction.INCOMING) > 3)
75                 crossTime += 2.0;
76
77              // cost function
78              double travelTime = current.cost.g + crossTime;
```

```
79
80              boolean toInsert = true;
81
82              // if in open list and g' < g
83              RouteNode routeHop = openSet.get(successorId); // ~O(1)
84              if(routeHop != null) {
85                 if(travelTime < routeHop.cost.g) {
86                    openSetHeap.remove(routeHop); // O(log n)
87                    openSet.remove(successorId); // ~O(1)
88                 } else
89                    toInsert = false;
90              }
91
92              // if in closed and g' < g
93              if(routeHop == null) {
94                 routeHop = closedSet.get(successorId);
95                 if (routeHop != null)
96                    if(travelTime < routeHop.cost.g) {
97                       closedSet.remove(successorId); // O(1)
98                       //toInsert = true;
99                    } else
100                      toInsert = false;
101             }
102
103             if(! toInsert)
104                continue;
105
106             if(routeHop == null)
107                routeHop = new RouteNode();
108
109             // We found a better path OR We first visited this node!
110             routeHop.cost.g = travelTime;
111             routeHop.cost.h = weight * heuristic(successor, end, maxspeed);
112             routeHop.node = successor;
113             routeHop.parent = current.getId();
114
115             openSet.put(successorId, routeHop);
116             openSetHeap.add(routeHop);
117          }
118       }
119
120       log.info(
121             "A*␣terminated␣in␣" + current.getId() +
122             "\topenSet␣size:␣" + openSet.size() + "\tclosedSet␣size:␣" +
                  ↪ closedSet.size()
123       );
124
125       if(current.getId() != getIdOf(end))
126          return Stream.<HopRecord>builder().build();
127
128       // Java non-sense
129       final RouteNode finalCurrent = current;
130       return StreamSupport.stream(Spliterators.spliteratorUnknownSize(new
             ↪ Iterator<>() {
131          long i = 0;
132          RouteNode step = finalCurrent;
133
134          @Override
135          public boolean hasNext() {
136             return step.getId() != getIdOf(start);
137          }
```

```
138
139         @Override
140         public HopRecord next() {
141             var record = new HopRecord();
142             record.index = i;
143             record.time = step.cost.g;
144             record.node = step.node;
145
146             step = closedSet.get(step.parent);
147             i++;
148
149             return record;
150         }
151     }, Spliterator.IMMUTABLE), false);
152 }
153
154 private double considerDisruptions(Node successor, double crossTime) {
155     var disruptionEdges = successor.getRelationships(Direction.OUTGOING,
          ↪ IS_DISRUPTED);
156     for (var disruptionEdge : disruptionEdges) {
157         Node disruption = disruptionEdge.getEndNode();
158         if (!disruption.hasProperty("severity")) {
159             log.warn("Disruption␣" + disruption.getElementId() + "␣has␣no␣
                  ↪ severity!!");
160             continue;
161         }
162
163         Double dw = disruptionWeights.get((String) disruption.getProperty("
              ↪ severity"));
164         if (dw == null) {
165             log.warn("Uknown␣severity␣" + disruption.getProperty("severity"));
166             continue;
167         }
168
169         if ((Boolean) disruption.getProperty("closed", false)) {
170             log.debug(
171                   "Closed␣road␣at␣" + disruption.getElementId() +
172                   "\tPoint␣" + successor.getElementId()
173             );
174
175             return crossTime * 25;
176         }
177
178         crossTime *= dw;
179     }
180
181     return crossTime;
182 }
183
184 private static double distance(Node a, Node b) {
185     PointValue posA = (PointValue) a.getProperty("coord");
186     PointValue posB = (PointValue) b.getProperty("coord");
187
188     final var calc = posA.getCoordinateReferenceSystem().getCalculator();
189
190     return calc.distance(posA, posB);
191 }
192
193 private static double heuristic(Node a, Node b, double maxspeed) {
194     // Maximum speed limit in UK: 70mph
195     return (distance(a, b) / (maxspeed + 5) * MPH_TO_MS);
```

```
196     }
197
198     private static double crossTime(Relationship way, Node a, Node b, double
            ↪ maxspeed) {
199         double distance = distance(a,b);
200         return Math.max( // We take the longest cross time (ie lower speed)
201                 distance / (Double) way.getProperty("maxspeed"),
202                 distance / (maxspeed * MPH_TO_MS)
203         );
204     }
205
206     private static long getIdOf(Node node) {
207         return (long) node.getProperty("id");
208     }
209
210     private static class Cost implements Comparable<Cost> {
211         public double g;
212         public double h;
213
214         public Cost(double g, double h) {
215             this.g = g;
216             this.h = h;
217         }
218
219         public Cost() {}
220
221         @Override
222         public int compareTo(Cost b) {
223             return Double.compare(g + h, b.g + b.h);
224         }
225     }
226
227     private static class RouteNode implements Comparable<RouteNode> {
228         public Cost cost;
229         public Node node;
230
231         public long parent;
232
233         public RouteNode(Cost cost, Node node) {
234             this(cost, node, 0L);
235         }
236
237         public RouteNode(Cost cost, Node node, long parent) {
238             this.cost = cost;
239             this.node = node;
240             this.parent = parent;
241         }
242
243         public RouteNode() {
244             this.cost = new Cost();
245         }
246
247         @Override
248         public int compareTo(RouteNode o) {
249             return cost.compareTo(o.cost);
250         }
251
252         public long getId() {
253             return getIdOf(this.node);
254         }
255     }
```

```
256
257     public static class HopRecord {
258        public long index;
259        public Node node;
260        public double time;
261
262     }
263  }
```

15.2 Point finding

To ensure that the user selects a reachable point in the network given the user's transportation mode, we use *connects* relationships between points in the graph to reduce the probability of selecting an unreachable point.

```
1  WITH point({latitude: $lat, longitude: $lng}) AS q
2  MATCH (p:Point)
3  MATCH (p)-[w:CONNECTS]->(r:Point)
4  WHERE point.distance(q, p.coord) < 100 AND
5  CASE
6    WHEN $type = 'foot' THEN w.crossTimeFoot <> Infinity
7    WHEN $type = 'bicycle' THEN w.crossTimeBicycle <> Infinity
8    WHEN $type = 'car' THEN w.crossTimeMotorVehicle <> Infinity
9  END
10 RETURN p, point.distance(q, p.coord)
11 ORDER BY point.distance(q, p.coord) LIMIT 1
```

Figure 15.2: Cypher query

For example, we in the user selects a pathway in a park and *motor vehicle* is selected as transportation mode, the query will return the node relative to the nearest road open to motor traffic.

As stated in the previous chapters, a restriction of access for a certain mode of transportation is represented in the graph as cross time of positive infinity.

The function *nearestNode* in Java has been implemented as show below:

```
1     public Point nearestNode(double lat, double lng){
2        return nearestNode(lat, lng, "car");
3     }
4
5     public Point nearestNode(double lat, double lng, String type){
6        try(var session = driver.session()){
7           var p = session.run(
8                 NEAREST_NODE.withParameters(
9                       parameters("lat",lat,
10                            "lng", lng, "type", type)));
11          if(!p.hasNext())
12             return null;
13
14          var record = p.single().get(0);
15
16          return new Point(
17                record.get("id").asLong(),
18                record.get("coord").asPoint().y(),
```

15.3 Create and update disruptions

Create a disruption 'd' and find all points 'p' that are within a specified radius. If necessary, create a relationship 'IS_DISRUPTED' between 'p' and 'd'.

```
1  MERGE (d:Disruption {id: $id})
2  SET d.centrum = point({latitude: $lat, longitude: $lon})
3  SET d.radius = $radius
4  SET d.severity = $severity
5  SET d.ttl = $ttl
6  SET d.category = $category SET d.subCategory = $subcategory
7  SET d.comment = $comment
8  SET d.update = $update SET d.updateTime = $updateTime
9  SET d.closed = $closed
10 WITH d
11 MATCH (p: Point)
12 WHERE point.distance(p.coord, d.centrum) <= d.radius
13 MERGE (p)-[:IS_DISRUPTED {severity: $severity}]->(d)
```

Figure 15.3: Cypher query

Chapter 16

Queries on Document database

In this chapter are presented the relevant queries for the MongoDB document database.

16.1 POIs in a certain area

Visualize the information of POIs in a given area

```
1     public PointOfInterest getPOI(String id) {
2        Bson match = eq("poiID", id);
3        return collection.find(match).first();
4     }
5
6     public List<PointOfInterest> selectPOIsInArea(double minLong,
7                                        double maxLong, double minLat,
8                                        double maxLat)
9     {
10       assert (minLong < maxLong);
11       assert (minLat < maxLat);
12
13       Polygon region = new Polygon(Arrays.asList(
14             new Position(minLong, minLat),
15             new Position(maxLong, minLat),
16             new Position(maxLong, maxLat),
17             new Position(minLong, maxLat),
18             new Position(minLong, minLat)
19       ));
```

The same query wrote in Mongo Query Language:

```
db.PointOfInterest.find(
{
  "coordinates": {
    $geoWithin: {
      $polygon: [
      [minLong, minLat],
      [maxLong, minLat],
      [maxLong, maxLat],
      [minLong, maxLat],
      [minLong, minLat]
      ]
    }
  }
}
)
```

Figure 16.1: POIs' MongoDB query

16.2 The heatmap

Build a heatmap of a certain class of disruption.

```
      return result;
   }

   /*
   Build a heatmap of a certain class of disruption
    */

   public Collection<Document> queryHeatmap(
         double lenLat,
         double lenLong,
         String classDisruption)
   {
      Bson match = match(eq("category", classDisruption));
      Bson computeBuckets = new Document("$project", new Document()
            .append("latB", new Document("$multiply", Arrays.asList(
                  new Document("$floor", new Document("$divide", Arrays.asList(
                     new Document("$arrayElemAt", Arrays.asList(
                           "$coordinates.coordinates",
                           1
                     )),
                     lenLat))),
                  lenLat
            )))
            .append("lngB", new Document("$multiply", Arrays.asList(
                  new Document("$floor", new Document("$divide", Arrays.asList(
                     new Document("$arrayElemAt", Arrays.asList(
                           "$coordinates.coordinates",
                           0
                     )),
                     lenLong))),
                  lenLong
            ))));
      Bson groupStage = Aggregates.group(
            new Document("latB", "$latB").append("lngB", "$lngB"),
            Accumulators.sum("count", 1)
      );
```

```
38
39        Bson project = project(fields(
40              excludeId(),
41              include("count"),
42              computed("latitude", "$_id.latB"),
43              computed("longitude", "$_id.lngB")
44        ));
45
46        // Create the pipeline
47        List<Bson> pipeline = Arrays.asList(
48              match, computeBuckets, groupStage, project
49        );
50        pipeline.forEach(bson -> System.out.println(bson.toBsonDocument()));
51
52
53        // Execute the aggregation
54        return collection.aggregate(pipeline).map(document -> {
55           var lat = document.getDouble("latitude");
56           var lon = document.getDouble("longitude");
57
58           return document;
59        }).into(new ArrayList<>());
60     }
61  }
```

The same query wrote in Mongo Query Language:

```
1  db.Disruption.aggregate([
2  { $match: { category: classDisruption } },
3  {
4    $project: {
5      latB: {
6        $multiply: [ {
7          $floor: {
8            $divide: [ {
9              $arrayElemAt:
10             [ "$coordinates.coordinates", 1 ]
11           },
12          lenLat ] } },
13        lenLat ] },
14     lngB: {
15       $multiply: [ {
16         $floor: {
17           $divide: [ {
18             $arrayElemAt: [
19             "$coordinates.coordinates", 0 ]
20           }, lenLong ] } },
21         lenLong ] }
22   }
23 },
24 {
25   $group: {
26     _id: { latB: "$latB", lngB: "$lngB" },
27     count: { $sum: 1 }
28   }
29 },
30 {
31   $project: {
32     count: 1,
33     latitude: "$_id.latB",
34     longitude: "$_id.lngB"
```

```
35    }
36  }
37  ])
```

16.3 The most common disruptions

Return a list witch contains the most common disruption in order to severity.

```
1     public Collection<Document> commonDisruptionInArea(
2           double minLong, double maxLong, double minLat, double maxLat) {
3        Polygon region = new Polygon(Arrays.asList(
4              new Position(minLong, minLat),
5              new Position(maxLong, minLat),
6              new Position(maxLong, maxLat),
7              new Position(minLong, maxLat),
8              new Position(minLong, minLat)
9        ));
10
11       Bson inSquare = geoWithin("coordinates", region);
12
13
14       // Create the group stage
15       Bson groupStage = Aggregates.group(
16             new Document("severity", "$severity").append("category", "$category
                 ↪ "),
17             Accumulators.sum("count", 1)
18       );
19       Bson groupStage2 = Aggregates.group(
20             "$_id.severity",
21             Accumulators.max("count", "$count"),
22             Accumulators.first("type", "$_id.category"),
23             Accumulators.first("severity", "$_id.severity")
24       );
25
26       // Create the sort stage
27       Bson sortStage = Aggregates.sort(Sorts.descending("count"));
28
29       // Projèct
30
31       Bson project = project(fields(excludeId(), include("severity",
32             "type", "count")));
33
34       // Combine the stages into a pipeline
35       List<Bson> pipeline = Arrays.asList(Aggregates.match(inSquare),
36             groupStage, groupStage2, sortStage, project);
37
38       // Execute the aggregation
39       Collection<Document> result = collection
40             .aggregate(pipeline).into(new ArrayList<>());
41
42       return result;
43    }
```

The same query wrote in Mongo Query Language:

```
1   db.Disruption.aggregate([
2   {
3     $match: {
4       coordinates: {
5         $geoWithin: {
```

```
        $polygon: [
        [minLong, minLat],
        [maxLong, minLat],
        [maxLong, maxLat],
        [minLong, maxLat],
        [minLong, minLat]
        ]
       }
      }
    }
  },
  {
    $group: {
      _id: { severity: "$severity", category: "$category" },
      count: { $sum: 1 }
    }
  },
  {
    $group: {
      _id: "$_id.severity",
      count: { $max: "$count" },
      type: { $first: "$_id.category" },
      severity: { $first: "$_id.severity" }
    }
  },
  {
    $sort: { count: -1 }
  },
  {
    $project: {
      _id: 0,
      severity: 1,
      type: 1,
      count: 1
    }
  }
  ])
```

16.4 Time series

Returns, for each hour of the day, the average number of disruptions that were active at that time of day. Optionally the user can filter by disruption class to produce a graph relative only the specified class.

```
   public List<LineGraphEntry> computeGraph(String category) {
      List<Bson> pipeline = new ArrayList<>();
      if(category != null)
         pipeline.add(Aggregates.match(Filters.eq("category", category)));


      pipeline.addAll(Arrays.asList(
      Aggregates.project(
    Projections.fields(
          new Document("start",
                new Document("$multiply",
                      Arrays.asList(
                            new Document("$floor",
                                  new Document("$divide",
                                        Arrays.asList(
                                  new Document("$toLong", "$start"),
```

```
17                                          3600000
18                                             )
19                                       )
20                                    ),
21                                    3600
22                                 )
23                           )
24                  ).append("end",
25                     new Document("$min",
26                           Arrays.asList(
27                                 new Document("$multiply",
28                                       Arrays.asList(
29                                       new Document("$ceil",
30                                          new Document("$divide",
31                                                Arrays.asList(
32                                             new Document("$toLong", "$end"),
33                                                   3600000
34                                                   )
35                                             )
36                                          ),
37                                          3600
38                                       )
39                                    ),
40                                 new Document("$multiply",
41                                    Arrays.asList(
42                                    new Document("$ceil",
43                                    new Document("$divide",
44                                          Arrays.asList(
45                                                new Date().getTime(),
46                                                   3600000
47                                             )
48                                          )
49                                          ),
50                                          3600
51                                       )
52                                    )
53                                 )
54                           )
55                  ),
56                  Projections.include("id"),
57                  Projections.include("category")
58            )
59      ),
60      Aggregates.project(
61            Projections.fields(
62                  Projections.computed("dates",
63                        new Document("$map",
64                              new Document("input",
65                                    new Document("$range",
66                              Arrays.asList("$start", "$end", 3600)
67                                    )
68                              ).append("as", "i").append("in", "$$i")
69                        )
70                  ),
71                  Projections.include("id"),
72                  Projections.include("category")
73            )
74
75               ),
76               Aggregates.unwind("$dates"),
77               Aggregates.project(
```

```
78                    Projections.fields(
79                          Projections.computed("date",
80                             new Document("$toDate",
81                                   new Document("$multiply",
82                                         Arrays.asList("$dates", 1000)
83                                   )
84                          )),
85                          Projections.include("id"),
86                          Projections.include("category")
87                    )
88              ),
89              Aggregates.project(
90                    Projections.fields(
91  Projections.computed("year", new Document("$year", "$date")),
92  Projections.computed("dayOfYear", new Document("$dayOfYear", "$date")),
93  Projections.computed("hour", new Document("$hour", "$date")),
94  Projections.include("id"),
95  Projections.include("category"
96                    )
97              )),
98              Aggregates.match(
99                    Filters.or(
100 Filters.ne("category", "Works"),
101 Filters.in("hour", Arrays.asList(7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18))
102                )
103             ),
104             Aggregates.group(
105                   Projections.fields(
106         Projections.computed("year", "$year"),
107         Projections.computed("dayOfYear", "$dayOfYear"),
108         Projections.computed("hour", "$hour")
109                   ),
110                   Accumulators.sum("count", 1L)
111             ),
112 Aggregates.group("$_id.hour", Accumulators.avg("count","$count")),
113 Aggregates.sort(Sorts.ascending("_id"))
114       ));
115
116       //pipeline.forEach(bson -> System.out.println(bson.toBsonDocument().
              ↪ toJson()));
117
118       return this.collection.aggregate(pipeline).into(new ArrayList<>());
119   }
```

The same query wrote in Mongo Query Language:

```
1  db.Disruption.aggregate([
2     /*{
3        $match: {subCategory: "Burst Water Main"}
4     },*/
5     {
6        $project: {
7           start: {$multiply : [{$floor: {$divide: [
8             {$toLong: "$start"}, 3600000]}}, 3600]},
9           // @TODO take min between $end and current_date
10          end: {$multiply : [{$ceil: {$divide:
11            [{$toLong: "$end"}, 3600000]}}, 3600]},
12          id: "$id",
13          category: "$category"
14       }
15    },
```

```
16      {
17         $project: {
18            dates: {
19               $map: {
20                  input: {$range: ["$start", "$end", 3600]},
21                  as: "i",
22                  in: "$$i"
23               }
24            },
25            id: "$id",
26            category: "$category"
27         }
28      },
29      {
30         $unwind: "$dates"
31      },
32      {
33         $project: {
34            date: {$toDate: {$multiply: ["$dates", 1000]}},
35            id: "$id",
36            category: "$category"
37         }
38      },
39      {
40         $project: {
41            year: {$year: "$date"},
42            dayOfYear: {$dayOfYear: "$date"},
43            hour: {$hour: "$date"},
44            id: "$id",
45            category: "$category"
46         }
47      },
48      {
49         $match: {$or: [
50               {
51                  category: {$ne: "Works"}
52               },
53               {
54                  hour: {$in: [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]}
55               }
56            ] }
57      },
58      {
59         $group: {
60            _id: {year: "$year", dayOfYear: "$dayOfYear", hour: "$hour"},
61            count: {
62               $count: {}
63            }
64         }
65      },
66      {
67         $group: {
68            _id: "$_id.hour",
69            count: {
70               $avg: "$count"
71            }
72         }
73      }
74   ])
```

16.5 Find a place

Returns a list of Points of Interest (POIs) whose names contain a case-insensitive substring specified by the user in the query.

The same query wrote in Mongo Query Language:

```
1  db.PointOfInterest.find(
2  { "name": {
3     $regex: name, $options: "i"
4    }
5  }
6  ).limit(20)
```

Figure 16.2: MongoDB query

Chapter 17

Neo4j indexes

To speed-up the queries on graph, two indexes on the *Point* nodes were added with the following DDL *Cypher* query:

```
1 CREATE POINT INDEX FOR (p:Point) ON (p.coord);
2 CREATE INDEX FOR (p:Point) ON (p.id);
```

resulting in two indexes, one **geo-spatial** and one **on the id**.

id	state	popPercent	type	entityType	labels	properties	provider
3	ONLINE	100.0	POINT	NODE	[Point]	[coord]	point-1.0
4	ONLINE	100.0	RANGE	NODE	[Point]	[id]	range-1.0

17.1 Geo-spatial index

Definition For point indexing, Neo4j uses space filling curves in 2D or 3D over an underlying generalized *B+Tree*. Points will be stored in up to four different trees, one for each of the four coordinate reference systems. This allows for both equality and range queries using exactly the same syntax and behavior as for other property types. If two range predicates are used, which define minimum and maximum points, this will effectively result in a bounding box query. In addition, queries using the `distance` function can, under the right conditions, also use the index.

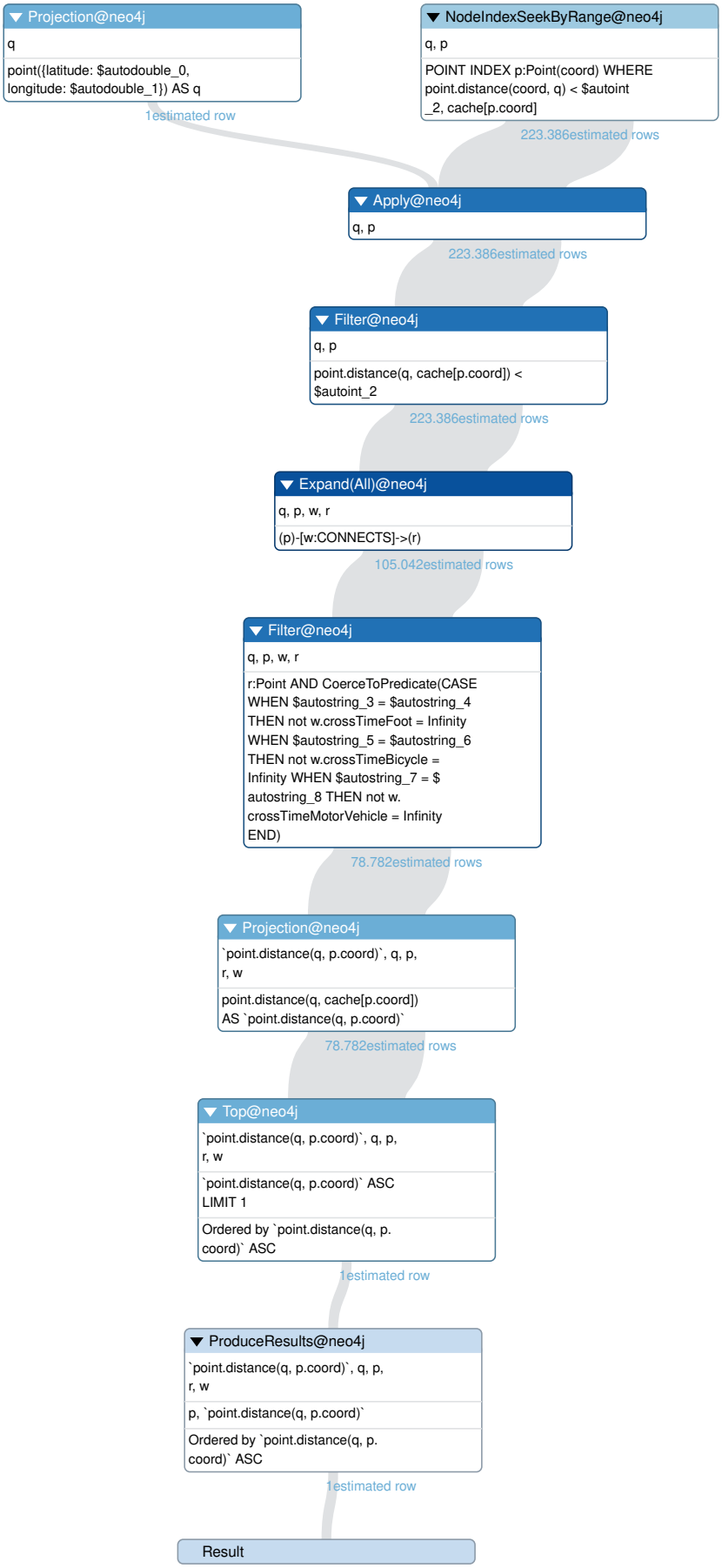
Usage We use the index mainly for the following queries:

- Locate the nearest point 15.2 given coordinates
- Insert operation of a disruption by *TIMS* 13.1, to select the roads effected by such disruption

For **both** queries the index is used to accelerate the look-up via the `point.distance()` built-in function. Here is how the query planner of the *DBMS* uses the index to find nearest point on the graph given a pair of coordinates as input:

Planner We can see how the entire domain is immediately filter in `NodeIndexSeekByRange@neo4j`, the first stage of the query, thus reducing the cardinality of the search domain from 7×10^6 to a more manageable 2×10^3.

The query plan was generate with the `EXPLAIN` Cypher *SQL* command.


Projection@neo4j
q
point({latitude: $autodouble_0, longitude: $autodouble_1}) AS q
1estimated row
NodeIndexSeekByRange@neo4j
q, p
POINT INDEX p:Point(coord) WHERE point.distance(coord, q) < $autoint _2, cache[p.coord]
223.386estimated rows
Apply@neo4j
q, p
223.386estimated rows
Filter@neo4j
q, p
point.distance(q, cache[p.coord]) < $autoint_2
223.386estimated rows
Expand(All)@neo4j
q, p, w, r
(p)-[w:CONNECTS]->(r)
105.042estimated rows
Filter@neo4j
q, p, w, r
r:Point AND CoerceToPredicate(CASE WHEN $autostring_3 = $autostring_4 THEN not w.crossTimeFoot = Infinity WHEN $autostring_5 = $autostring_6 THEN not w.crossTimeBicycle = Infinity WHEN $autostring_7 = $ autostring_8 THEN not w. crossTimeMotorVehicle = Infinity END)
78.782estimated rows
Projection@neo4j
`point.distance(q, p.coord)`, q, p, r, w
point.distance(q, cache[p.coord]) AS `point.distance(q, p.coord)`
78.782estimated rows
Top@neo4j
`point.distance(q, p.coord)`, q, p, r, w
`point.distance(q, p.coord)` ASC LIMIT 1
Ordered by `point.distance(q, p. coord)` ASC
1estimated row
ProduceResults@neo4j
`point.distance(q, p.coord)`, q, p, r, w
p, `point.distance(q, p.coord)`
Ordered by `point.distance(q, p. coord)` ASC
1estimated row
Result

Benchmark To benchmark the query 15.2 we ran a thousand times the query and took the *minimum*, *average* and *maximum* execution time.

```
driver = GraphDatabase.driver("bolt://localhost:7687", auth=("neo4j", "password
    ↪ "))
session = driver.session()

def make_query(typeV, lat, lng):
  session.execute_read(query, typeV, lat, lng)

def query(tx, typeV, lat, lng):
  res = tx.run(
"""
WITH point({latitude: $lat, longitude: $lng}) AS q
MATCH (p:Point)
MATCH (p)-[w:CONNECTS]->(r:Point)
WHERE point.distance(q, p.coord) < 100 AND
CASE
  WHEN $typeV = 'foot' THEN w.crossTimeFoot <> Infinity
  WHEN $typeV = 'bicycle' THEN w.crossTimeBicycle <> Infinity
  WHEN $typeV = 'car' THEN w.crossTimeMotorVehicle <> Infinity
END
RETURN p, point.distance(q, p.coord)
ORDER BY point.distance(q, p.coord) LIMIT 1
""", typeV=typeV, lat=lat, lng=lng)

MAX_LAT = 51.7314463;
MIN_LAT = 51.2268448;
MAX_LON = 0.399670;
MIN_LON = -0.6125035;

typeVs = ["foot", "bicycle", "car"]

mint = 0xffffffff
maxt = 0
sumt = 0

for i in range(1, 1000):
  lat = MIN_LAT + (MAX_LAT - MIN_LAT)*random.uniform(0, 1)
  lng = MIN_LON + (MAX_LON - MIN_LON)*random.uniform(0, 1)

  start_time = time.time()

  make_query(typeVs[i%3], lat, lng)

  ext = time.time() - start_time
  mint = min(ext, mint)
  maxt = max(ext, maxt)
  sumt = sumt + ext

print(mint, maxt, (sumt/1000))
```

We obtained the following results, figures in *seconds*:

index	min	max	avg
yes	0.001	0.307	0.141
no	4.909	5.424	4.932

Conclusion We obtain a speed-up of circa forty times when using the geo-spatial index!

Chapter 18

MongoDB indexes

18.1 Point of Interest

To speed-up the queries on the document database a geo-spatial index was added on the *PointOfInterest* collection.

key	name
{"coordinates": "2dsphere"}	coordinates_2dsphere

Definition A `2dsphere` index supports queries that calculate geometries on an earth-like sphere. `2dsphere` index supports all MongoDB geo-spatial queries: queries for *inclusion*, *intersection* and *proximity*.

Usage

Planner We can see that MongoDB query engine use the index to speed-up the calls to the operator `$geoWithin`, in the stage called `IXSCAN`.

The planner estimates a reduction in the number of comparisons from circa 10^3 to circa 10.

```
{
  "winningPlan": {
    "stage": "FETCH",
    "filter": {
      "coordinates": {
        "$geoWithin": {
          "$geometry": {
            "type": "Polygon",
            "coordinates": [...]
          }
        }
      }
    },
    "inputStage": {
      "stage": "IXSCAN",
      "keyPattern": {
        "coordinates": "2dsphere"
      },
      "indexName": "coordinates_2dsphere",
      "isMultiKey": false,
      "multiKeyPaths": {
        "coordinates": [
        ]
      },
      "isUnique": false,
      "isSparse": false,
```

```
      "isPartial": false,
      "indexVersion": 2,
      "direction": "forward",
      "indexBounds": {
        "coordinates": [...]
      }
    }
  }
}
```

Benchmark To benchmark the query 13.4 we ran one-hundred-fifty times the query and took the *minimum*, *average* and *maximum* execution time.

```
const MAX_LAT = 51.7314463;
const MIN_LAT = 51.2268448;
const MAX_LON = 0.399670;
const MIN_LON = -0.6125035;

function generateRandomBox() {
   const result = {};

   result.minLong = MIN_LON + (MAX_LON - MIN_LON)*Math.random();
   result.minLat = MIN_LAT + (MAX_LAT - MIN_LAT)*Math.random();

   result.maxLong = result.minLong + 0.0086530;
   result.maxLat = result.minLat + 0.0045566;

   return result;
}

function makeQuery(box) {
   const result = db.PointOfInterest.find(
      {
         "coordinates": {
            $geoWithin: { $geometry: {
                  type : "Polygon" ,
                  coordinates: [[
                     [box.minLong, box.minLat],
                     [box.maxLong, box.minLat],
                     [box.maxLong, box.maxLat],
                     [box.minLong, box.maxLat],
                     [box.minLong, box.minLat]
                  ]]
               }
            }
         }
      });

   return result.explain("executionStats").executionStats.executionTimeMillis
}

function Result(index) {
   this.index = index;
   this.mint = 99999.0;
   this.maxt = 0.0;
   this.avgt = 0.0;

};

function bench(iter, result) {
   for(let i = 0; i < iter; i++) {
```

```
49        const t = makeQuery(generateRandomBox());
50        result.mint = Math.min(result.mint, t);
51        result.maxt = Math.max(result.maxt, t);
52        result.avgt += t;
53    }
54
55    result.avgt = result.avgt / iter;
56 }
57
58 const results = [];
59
60 db.PointOfInterest.dropIndex("coordinates_2dsphere");
61 const noIndexs = new Result(false);
62 bench(150, noIndexs);
63
64 // Create index
65 db.PointOfInterest.createIndex( { coordinates : "2dsphere" });
66 const yesIndexs = new Result(true);
67 bench(150, yesIndexs);
68
69 // Return to intellj
70 results.push(noIndexs, yesIndexs);
71 results;
```

We obtained the following results, figures in *milliseconds*:

index	min	max	avg
no	13	15	13.58
yes	0	4	0.0333

Conclusion We obtain an impressive speed-up. Unfortunately the size of the data is not large enough to make a meaningful comparison as *mongosh* gives executions in milliseconds as integer numbers, thus making fast queries appear as having zero as execution time.

Chapter 19

TIMS client

19.1 Implementation

The driver's implementation can be found in the class `driver.tims.RoadDisruptionUpdate`. Here's a small snapshot of the source code where the *Gson* library parses the incoming *JSON*:

```
1    private static ProcessResult process(InputStreamReader fs, ProcessResult
         ↪ last, Date t) throws Exception {
2       Type collectionType = new TypeToken<ArrayList<RoadDisruptionUpdate>>() {
3       }.getType();
4       Collection<RoadDisruptionUpdate> updates = new GsonBuilder()
5             .registerTypeAdapterFactory(new GeometryAdapterFactory())
6             .registerTypeAdapter(Street.Segment.class, new SegmentCodec())
7             .create().fromJson(fs, collectionType);
8
9       HashSet<String> explored = new HashSet<>();
10
11      if (updates == null || updates.isEmpty())
12         return null;
13
14      if (t.before(last.time))
15         throw new RuntimeException(
16               "I can only go forward in time! Current file was at " + t + " I
                  ↪ can only go after " + last.time);
17
18      HashMap<String, londonSafeTravel.schema.graph.Disruption>
            ↪ activeDisInGraph = new HashMap<>(150);
19      List<londonSafeTravel.schema.graph.Disruption> toWrite = new ArrayList
            ↪ <>();
20      disruptionDAOGraph.findDisruption().forEach(disruption -> {
21         activeDisInGraph.put(disruption.id, disruption);
22      });
23
24      System.err.println("Ready to list!");
25      updates.forEach(roadDisruptionUpdate -> {
26         // Add this disruption to the explored set
```

Then the program continues with the appropriate *CRUD* operations for each disruption update...

19.2 Configuration

User The TIMS driver client is a user of our application, its mission is to retrieve the current disruption from *Transport for London*'s *API*. It automatically executed by a *SystemD* timer every ten minutes.

SystemD The unit file describing the *one-shot* service to launch the driver:

```
[Unit]
Description=Retrive data from the TfL API and pours them into lsmdb's application
Wants=tfl.timer

[Service]
Type=oneshot
User=duke
Group=duke
ExecStart=/home/duke/service/download.sh

[Install]
```

The unit file describing the periodic timer:

```
[Unit]
Description=tfl's timer
Requires=tfl.service

[Timer]
Unit=tfl.service
OnBootSec=10min
OnUnitActiveSec=10min

[Install]
WantedBy=timers.target
```

Launcher script The following shell script downloads the data from the *API* and feeds them to the driver

```
#!/bin/bash

# Insert key here
KEY=

pushd $( dirname -- "$0"; )/

wget https://api.tfl.gov.uk/Road/all/Disruption?app_key=$KEY -O - | java -cp libCommon-0.0.3-jar-with-dependencies.jar londonSafeTravel.driver.tims.RoadDisruptionUpdate

popd
```

Part V

Conclusion

19.3 The Benefits of Using Graph and Document Databases in Transportation Systems

In addition to meeting all objectives, the use of a graph database in our application has also brought several advantages. It has provided us with a highly flexible and scalable data storage solution, allowing us to easily add and modify data as needed. The graph structure of the database allows for efficient querying and indexing of data, enabling quick and accurate retrieval of information. This is particularly useful in the context of routing, as it allows for real-time updates to the routing information and the ability to quickly find alternative routes for users in the event of a disruption. We also used a document database to store the history of disruptions and Points of Interest (POIs).

By using a document database, we were able to store detailed information about each disruption and POI, including timestamps, location data, and any other relevant details. This allows us to maintain a historical record of disruptions and POIs, and provides us with valuable data for analyzing past disruptions and identifying patterns.

Furthermore, the use of a graph database also enables more advanced analytics and data visualization. The ability to create relationships between different data points and analyze them as a network provides a deeper understanding of the data and allows for more accurate predictions and recommendations. This can be particularly beneficial for the statistician, as it would enable them to make more informed decisions and proposals for improving traffic in London.

The use of both a graph database and a document database in our application allowed us to efficiently store and manage a wide variety of data, providing us with a comprehensive view of the transportation system and enabling more accurate predictions and recommendations for improving traffic in London.

19.4 Possible expansions in the future

In conclusion, the use of a graph and document database in our application has not only allowed us to meet our objectives but also brought several advantages such as efficient querying, advanced analytics, and scalability. It has opened up the possibility for more advanced functionalities to be implemented in the future, such as incorporating data on public transportation and the disruptions that it can cause, providing more comprehensive information and enabling more accurate predictions and recommendations for improving traffic in London.

19.5 Possible query on public trasportations

For each class of public transportation disruption, find the top 3 lines that are most affected.

```
db.Disruption.aggregate([
{
  $match: {
    typeDisruption: "PUBLIC_TRANSPORT"
  }
},
{
  $group: {
    _id: "$terminatedDisruption.category",
    count: { $sum: 1 },
    line: { $first: "$routes.line" }
  }
},
{
  $sort: { count: -1 }
},
{
  $limit: 3
}
])
```

Part VI

Manual

Appendix A

User

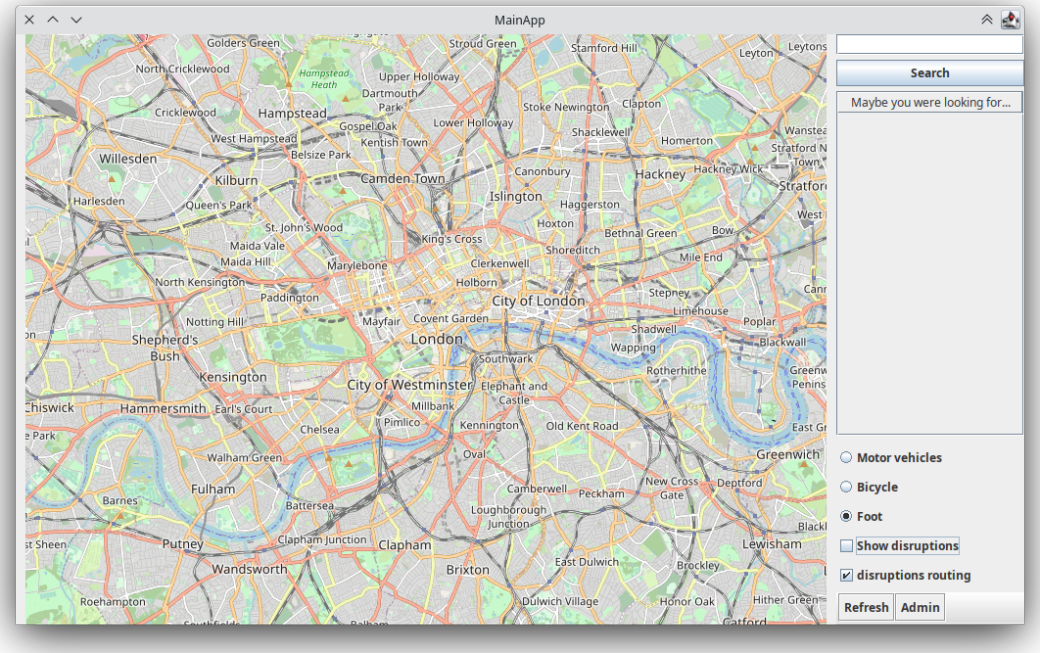


Figure A.1: Application just after launch

Introduction When launched the application automatically connects to the remote server and shows a map centered in London. The user can navigate the map with the typical *drag 'n drop* that is common with those kind of applications and he can change the magnifying level with the scrolling wheel of the mouse.

A.1 Point of interests

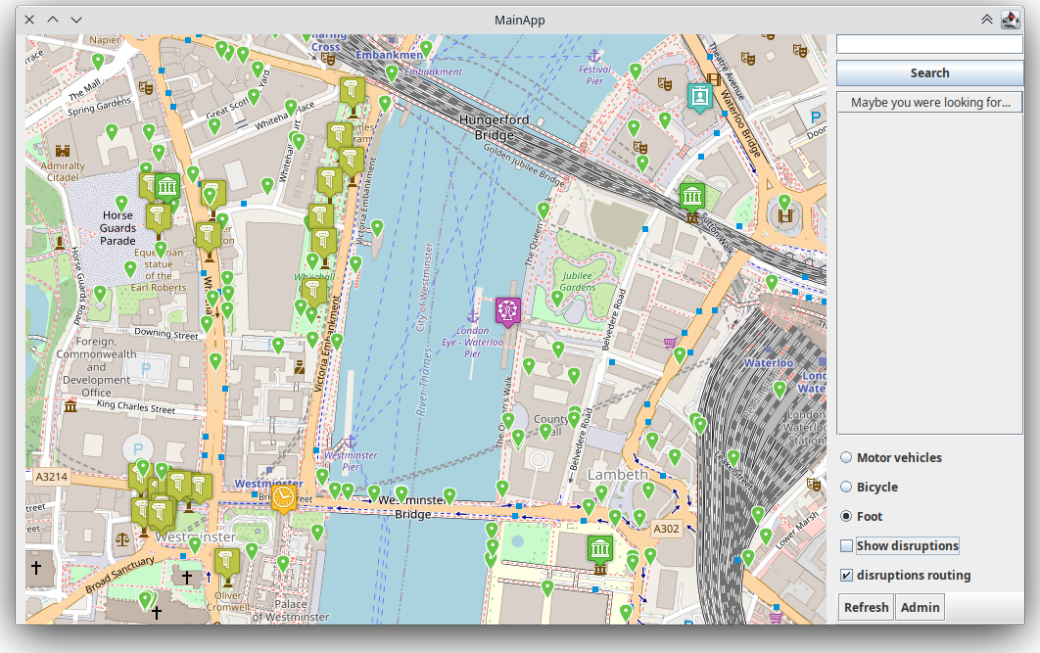


Figure A.2: Point of Interests near Westminster

Appearance The *Point of Interest*s are being shown on the map only when an appropriate level of zoom is reached to avoid cluttering the map.

Information The user can access more information about a certain *Point of Interest* by left-clicking on it. The application will then open an additional dialog with relevant information as seen in A.3

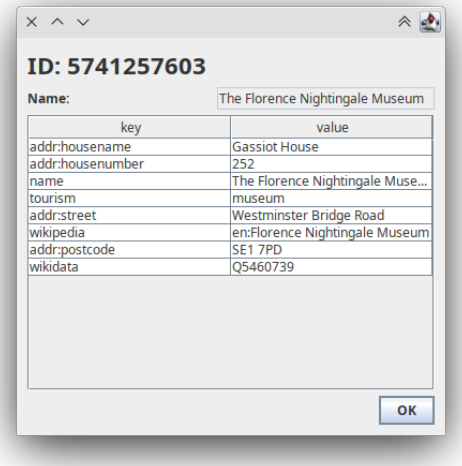


Figure A.3: Informations regarding the "The Florence Nightingale Museum"

A.2 Disruptions

Appearance The user can make the currently active disruptions appear on the map by ticking the option called *Show disruptions* located in the right panel. If needed the user can also force an update request by clicking on the button *Refresh*.

The disruptions are color-coded according to their severity:

1. **Grey** for disruptions with severity *minimal*
2. **Field drab** for disruptions with severity *moderate*
3. **Orange** for disruptions with severity *serious*
4. **Red** for disruptions with severity *severe*

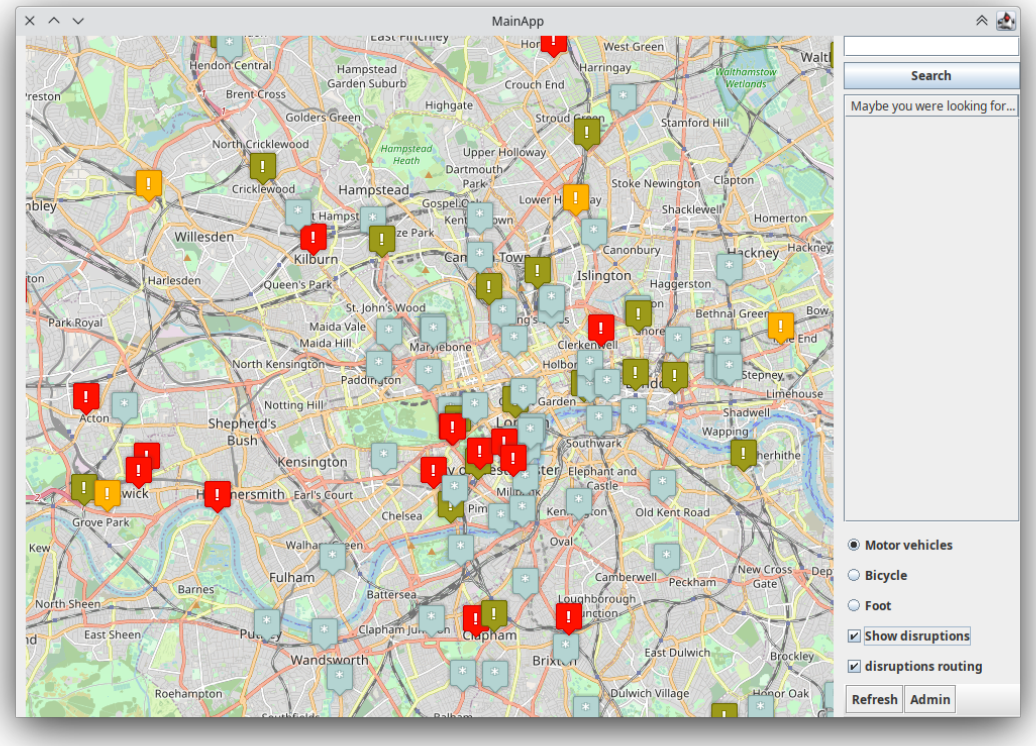


Figure A.4: Active disruptions in the afternoon of Janury 19th

Information The user can access more information about a certain *disruption* by left-clicking on it. The application will then open an additional dialog with relevant information as seen in A.5

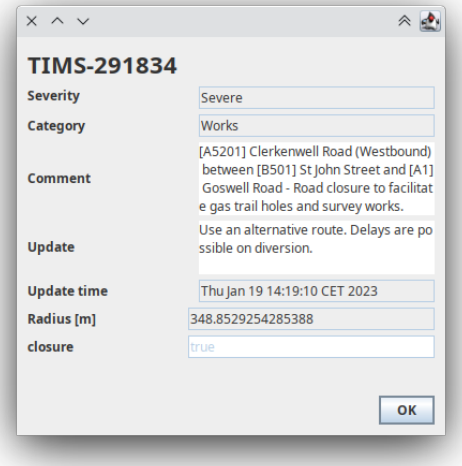


Figure A.5: Information regarding disruption *TIMS-291834*

A.3 Routing

Disruptions To make the routing consider delays caused by disruptions, tick the option called *disruption routing*.

Mode The routing algorithm support three modes of transport: on foot, by bicycle or by car; select the appropriate option from the right panel.

Destination To route between two points, right-click on the departure point on the map then right click on the destination point on the map; then the routing will begin and the result will be rendered on the map as soon as it is available.

(a) by car (b) by bicycle (c) on foot

Figure A.6: Comparison between the three modes

Results The path is rendered on the map as a red line, the estimated travel time is displayed in the right panel.

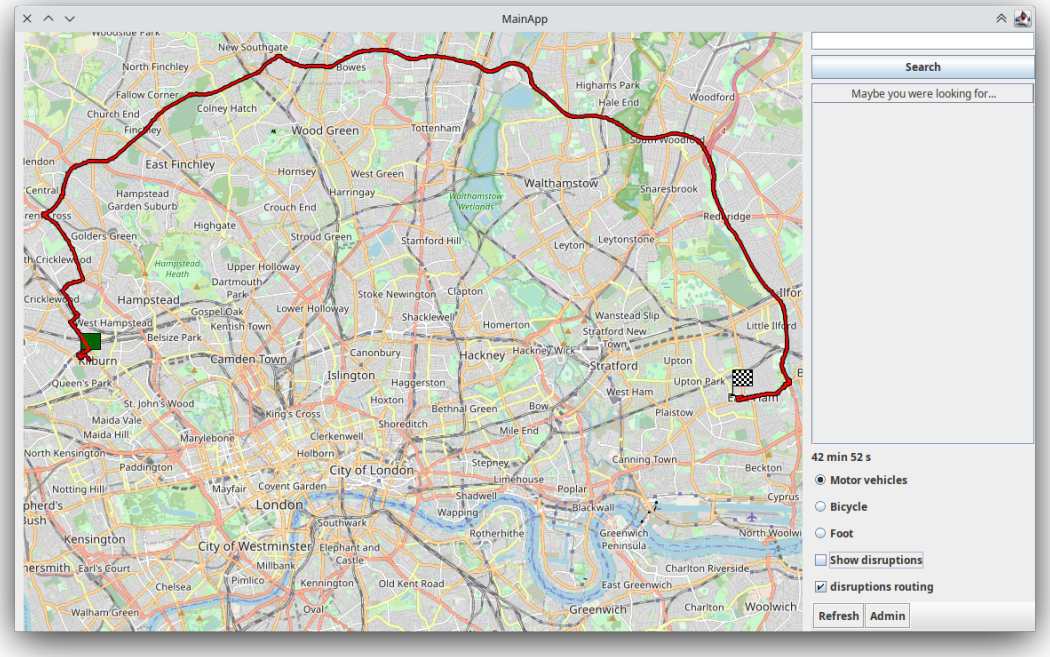
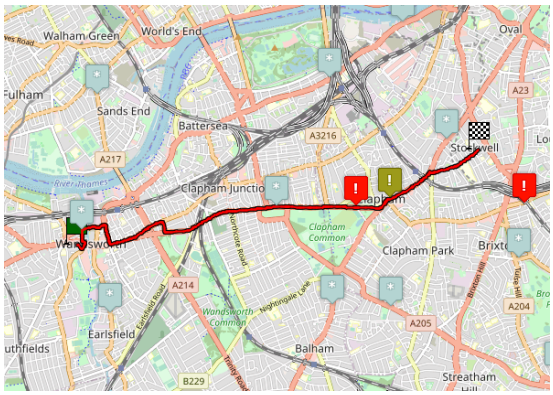


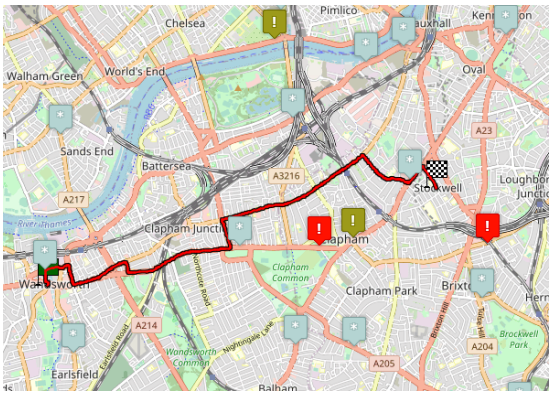
Figure A.7: Routing between Kilburn and East Ham. We can see how the routing procedure prefers faster motor-only roads when traveling by car

Time estimation The travel time estimations considers several factors such as

- The maximum speed for the given mode
- If in a vehicle, the speed limit
- If in a motor vehicle, the road class
- If enabled, the disruptions' severity



(a) disruptions off



(b) disruptions on

Figure A.8: Routing and disruptions

A.4 Search

It is possible to search on the map for *Point of Interest* by keyword.

Search To perform a search the user has to input his query in the*Search* input box in the right panel and then click on the *Search* button.

Result The map is centered on the most relevant *Point of Interest*, other results are shown in a list in the right panel, below the search button.

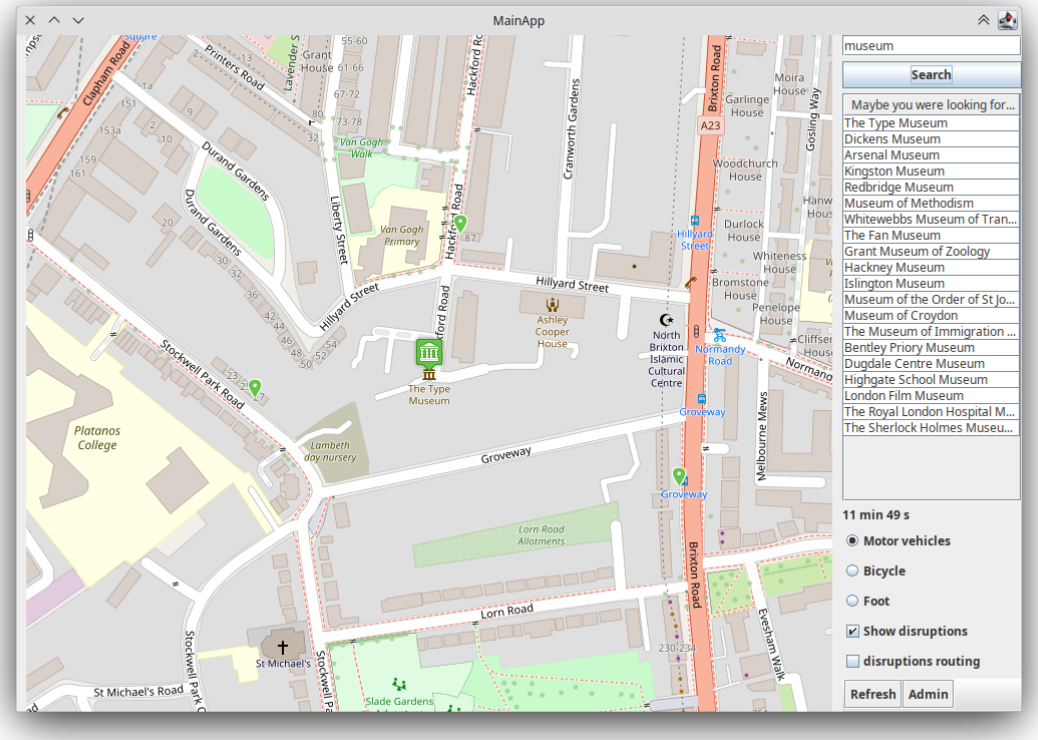


Figure A.9: Query results for *Museum*

Appendix B

Statistician

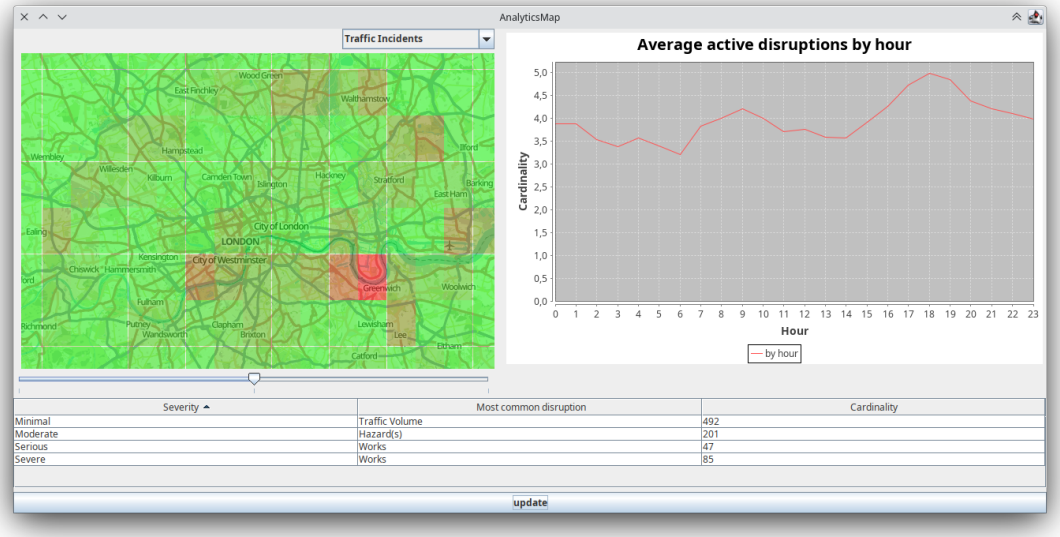


Figure B.1

Access The authorized personnel can access the administrative window by clicking on the *Admin* button and entering his credentials in the dialog. In the demo application is it possible to log-in with the following details:

```
Username: admin
Password: admin
```

Layout The analytic window is divided in three main panels, one for each analytic provided by the application.

B.1 Heat-map

The heat-map is located in the top-left pane of the window. It shows a heat-map relative to the selected disruption; the green color represents an area where a the class of disruption never happened, the red color represents an area where most of the disruptions happened.

Category To select the *category of disruption* to analyze, select it from the drop-down list above the map.

Precision To applications provides three levels of precision when computing the heat-map, to change it move the slider below the map; to the left is more precise, that is to render smaller squares, and to the right is less precise.

Movements As for the user's map, it is possible to move around via *drag 'n drop*.

B.2 Disruptions' time series

The time series is located on the top-right pane of the window. It shows how common was a disruption in a certain hour of the day.

Category The category is the same as the one selected for the heat-map.

B.3 Common disruptions in an area

The result of this analytic are presented in the table located in the bottom pane of the window. It shows, for each category, how many disruptions happened in the selected area, and what was the most common sub-category for each one.

Boundaries To select the boundaries move the map around and adjust its zoom, unlike the others, to update the result press the *update button*.