London Safe Travel

Dario Pagani, 585281
Federico Frati, 596237
Ricky Marinsalda, 585094

January 20, 2023

Contents

I Introduction **2**

II Feasibility **4**

1 Data sources **5**
- 1.1 Transport for London . . . 5
- 1.2 OpenStreetMap . . . 5

III Design **7**

2 Main Actors **9**

3 Functional requirements **10**

4 CAP Theorem issue **11**

5 Use cases **12**

6 Data model **13**
- 6.1 Document database . . . 13
- 6.2 Graph database . . . 13
- 6.3 Key-value database . . . 13

7 Distributed database **14**

8 Overall platform architecture **15**

9 Framework used **16**

IV Implementation **17**

10 Queries on graph database **18**
- 10.1 Routing . . . 18
- 10.2 Point finding . . . 22

V Conclusion **24**

VI Manual **25**

A User **26**

B Statistician **27**

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 three sources and two organizations:

- Transport for London TIMS
- Transport for London public disruptions API
- 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 TLF's JSON API with a simple shell script that was being executed automatically by a SystemD timer every ten minutes.

schema The main challenge was to find a "schema" for the document database to store in an unified way, that is to share the same MongoDB collection, between road disruptions and public transportation disruptions; as it could be useful for certain operations to have an unified view of such data.

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 iterate over several possible schemas for the graph database and over many different ways to import the raw data into it.

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

Figure 1.1: Example of intersections

Part III

Design

bla bla bla

Chapter 2

Main Actors

Chapter 3

Functional requirements

Chapter 4

CAP Theorem issue

Chapter 5

Use cases

Chapter 6

Data model

6.1 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.

6.2 Graph database

DBMS Our choice for the *database management system* to handle the graph database was *Neo4j*, since it is ???

6.3 Key-value database

DBMS Our choice for the *database management system* to handle the graph database was *Rédis*, since it provides excellent performance for our use case, that is to act as a cache for most frequent users' queries.

Chapter 7

Distributed database

Chapter 8

Overall platform architecture

Chapter 9

Framework used

Part IV

Implementation

Chapter 10

Queries on graph database

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

10.1 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 10.1: 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("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);
    HashMap<Long, RouteNode> closedSet = new HashMap<>(0xfeee, 0.666f); // Trust the Scienc

    var startNode = new RouteNode(new Cost(0, weight * heuristic(start, end, maxspeed)), st
    openSetHeap.add(startNode);
    openSet.put(startNode.getId(), startNode);
```

```

        log.info("Starting route from " + getIdOf(start) + "\tto " + getIdOf(end));

        RouteNode current = null;
        while(! openSet.isEmpty()) {
            assert (openSet.size() == openSetHeap.size());

            // Get current node
            current = openSetHeap.pollFirst(); // O(log n)
            openSet.remove(current.getId()); // ~O(1)

            // Move from open to closed
            closedSet.put(current.getId(), current); // ~O(1)

            // Found the solution HALT!
            if(current.getId() == getIdOf(end))
                break;

            var hops = current.node.getRelationships(Direction.OUTGOING, CONNECTS); // O(1), I

            for(var way : hops) {
                // NO ENTRY IN HERE!
                final double crossTimeStored = (Double) way.getProperty(crossTimeField);
                if(crossTimeStored == Double.POSITIVE_INFINITY)
                    continue;

                // Successor info
                Node successor = way.getEndNode();
                final long successorId = getIdOf(successor);

                // cost for this edge
                double crossTime = crossTime(way, current.node, successor, maxspeed);

                // Look for disruption
                var disruptionEdges = successor.getRelationships(Direction.OUTGOING, IS_DISRUPT
                for(var disruptionEdge : disruptionEdges) {
                    Node disruption = disruptionEdge.getEndNode();
                    if(! disruption.hasProperty("severity")) {
                        log.warn("Disruption " + disruption.getElementId() + " has no severity!
                        continue;
                    }

                    Double dw = disruptionWeights.get((String)disruption.getProperty("severity"
                    if(dw == null) {
                        log.warn("Uknown severity " + disruption.getProperty("severity"));
                        continue;
                    }

                    crossTime *= dw;
                }

                // Adjust for minor roads
                if(crossTimeField.equals("crossTimeMotorVehicle")) {
                    Double factor = timeWeights.get((String)way.getProperty("class"));
                    if(factor != null)
                        crossTime *= factor;
                }

                // If big intersection, add 5 seconds
                if(successor.getDegree(CONNECTS, Direction.INCOMING) > 3)
                    crossTime += 2.0;
```

```
                // cost function
                double travelTime = current.cost.g + crossTime;

                boolean toInsert = true;

                // if in open list and g’ < g
                RouteNode routeHop = openSet.get(successorId); // ~O(1)
                if(routeHop != null) {
                    if(travelTime < routeHop.cost.g) {
                        openSetHeap.remove(routeHop); // O(log n)
                        openSet.remove(successorId); // ~O(1)
                    } else
                        toInsert = false;
                }

                // if in closed and g’ < g
                if(routeHop == null) {
                    routeHop = closedSet.get(successorId);
                    if (routeHop != null)
                        if(travelTime < routeHop.cost.g) {
                            closedSet.remove(successorId); // O(1)
                            //toInsert = true;
                        } else
                            toInsert = false;
                }

                if(! toInsert)
                    continue;

                if(routeHop == null)
                    routeHop = new RouteNode();

                // We found a better path OR We first visited this node!
                routeHop.cost.g = travelTime;
                routeHop.cost.h = weight * heuristic(successor, end, maxspeed);
                routeHop.node = successor;
                routeHop.parent = current.getId();

                openSet.put(successorId, routeHop);
                openSetHeap.add(routeHop);
            }
        }

        log.info(
                "A* terminated in " + current.getId() +
                "\topenSet size: " + openSet.size() + "\tclosedSet size: " + closedSet.size()
        );

        if(current.getId() != getIdOf(end))
            return Stream.<HopRecord>builder().build();

        // Java non-sense
        final RouteNode finalCurrent = current;
        return StreamSupport.stream(Spliterators.spliteratorUnknownSize(new Iterator<>() {
            long i = 0;
            RouteNode step = finalCurrent;

            @Override
            public boolean hasNext() {
                return step.getId() != getIdOf(start);
```

```java
            }

            @Override
            public HopRecord next() {
                var record = new HopRecord();
                record.index = i;
                record.time = step.cost.g;
                record.node = step.node;

                step = closedSet.get(step.parent);
                i++;

                return record;
            }
        }, Spliterator.IMMUTABLE), false);
    }

    private static double distance(Node a, Node b) {
        PointValue posA = (PointValue) a.getProperty("coord");
        PointValue posB = (PointValue) b.getProperty("coord");

        final var calc = posA.getCoordinateReferenceSystem().getCalculator();

        return calc.distance(posA, posB);
    }

    private static double heuristic(Node a, Node b, double maxspeed) {
        // Maximum speed limit in UK: 70mph
        return (distance(a, b) / (maxspeed + 15) * MPH_TO_MS);
    }

    private static double crossTime(Relationship way, Node a, Node b, double maxspeed) {
        double distance = distance(a,b);
        return Math.max( // We take the longest cross time (ie lower speed)
                distance / (Double) way.getProperty("maxspeed"),
                distance / (maxspeed * MPH_TO_MS)
        );
    }

    private static long getIdOf(Node node) {
        return (long) node.getProperty("id");
    }

    private static class Cost implements Comparable<Cost> {
        public double g;
        public double h;

        public Cost(double g, double h) {
            this.g = g;
            this.h = h;
        }

        public Cost() {

        }

        @Override
        public int compareTo(Cost b) {
            return Double.compare(g + h, b.g + b.h);
        }
    }
```

```

    private static class RouteNode implements Comparable<RouteNode> {
        public Cost cost;
        public Node node;

        public long parent;

        public RouteNode(Cost cost, Node node) {
            this(cost, node, 0L);
        }

        public RouteNode(Cost cost, Node node, long parent) {
            this.cost = cost;
            this.node = node;
            this.parent = parent;
        }

        public RouteNode() {
            this.cost = new Cost();
        }

        @Override
        public int compareTo(RouteNode o) {
            return cost.compareTo(o.cost);
        }

        public long getId() {
            return getIdOf(this.node);
        }
    }

    public static class HopRecord {
        public long index;
        public Node node;
        public double time;

    }
}
```

10.2 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.

```
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 $type = 'foot' THEN w.crossTimeFoot <> Infinity
  WHEN $type = 'bicycle' THEN w.crossTimeBicycle <> Infinity
  WHEN $type = 'car' THEN w.crossTimeMotorVehicle <> Infinity
END
RETURN p, point.distance(q, p.coord)
ORDER BY point.distance(q, p.coord) LIMIT 1
```

Figure 10.2: 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.

Part V

Conclusion

Part VI

Manual

Appendix A

User

Appendix B

Statistician