Public Transit Route Planning Through Lightweight Linked Data Interfaces

Pieter Colpaert[(✉)], Ruben Verborgh, and Erik Mannens

Ghent University – imec – Internet and Data Lab, Ghent, Belgium
{pieter.colpaert,ruben.verborgh,erik.mannens}@ugent.be

Abstract. While some public transit data publishers only provide a data dump – which only few reusers can afford to integrate within their applications – others provide a use case limiting origin-destination route planning API. The Linked Connections framework instead introduces a hypermedia API, over which the extendable base route planning algorithm "Connections Scan Algorithm" can be implemented. We compare the CPU usage and query execution time of a traditional server-side route planner with the CPU time and query execution time of a Linked Connections interface by evaluating query mixes with increasing load. We found that, at the expense of a higher bandwidth consumption, more queries can be answered using the same hardware with the Linked Connections server interface than with an origin-destination API, thanks to an average cache hit rate of 78%. The findings from this research show a cost-efficient way of publishing transport data that can bring federated public transit route planning at the fingertips of anyone.

Keywords: Linked data · Public transport · Route planning · Open data

1 Introduction

The way travelers want route planning advice is diverse: from finding journeys that are accessible with a certain disability [2], to taking into account whether the traveler owns a (foldable) bike, a car or a public transit subscription, or even calculating journeys with the nicest pictures on social network sites [8]. However, when presented with a traditional route planning HTTP API taking origin-destination queries, developers of e.g. traveling tools are left with no flexibility to calculate journeys other than the functions provided by the server. As a consequence, developers that can afford a larger server infrastructure, integrate data dumps of the timetables (and their real-time updates), into their own system. This way, they are in full control of the algorithm, allowing them to calculate journeys in their own manner, across data sources from multiple authorities.

When publishing departure-arrival pairs (*connections*) in chronologically ordered pages – as demonstrated in earlier work [4] – route planning can be

executed at data integration time by the user agent rather than by the data publishing infrastructure. This way, the *Linked Connections (*LC*) framework*[1] allows for a richer web publishing and querying ecosystem within public transit route planners. It lowers the cost for reusers to start prototyping – also federated and multimodal – route planners over multiple data sources. Furthermore, other sources may give more information about a certain specific vehicle that may be of interest to this user agent. It is not exceptional that route planners also take into account properties such as fares [5], wheelchair accessibility [2] or criminality statistics. In this paper, we test our hypothesis that this way of publishing is more lightweight: how does our information system scale under more data reuse compared to origin-destination APIs?

The paper is structured in a traditional way, first describing state of the art, then introducing more details on the Linked Connections framework, to then describe the evaluation's design, made entirely reproducible, and discuss our open-source implementation. Finally, we elaborate on the results and conclusion.

2   State of the Art

The General Transit Feed Specification (GTFS)[2] is a framework for exchanging data from a public transit agency to third parties. It is – at the time of writing – the de-facto standard for describing and exchanging transit schedules. It describes the headers of several CSV files combined in a ZIP-file. Using a calendar and calendar exceptions, both periodic as aperiodic transit schedules can be described. GTFS also describes the geographic shape of trips, fare zones, accessibility, information about an agency, and so forth. We provided URIs to the terms in the GTFS specification through the Linked GTFS[3] vocabulary. Reusing these GTFS files, route planners exist in software as a service platforms such as Navitia.io, in end-user applications such as CityMapper, Ally or Google Maps, or as open-source software, such as Open Trip Planner or Bliksemlabs RRRR. It is also possible a that an agency, such as the Dutch railway company, the SNCB or the Deutsche Bahn, expose a route planner over HTTP themselves.

Data dumps and query APIs can be seen as two extremes on the *Linked Data Fragments* (LDF)[4] axis. Triple Pattern Fragments – similarly to Linked Connections [4] – already defined a way to publish an RDF dataset only through its triple patterns [10]. This moves the evaluation of all Basic Graph Patterns – and by extension SPARQL queries – to the client, reducing CPU load on the server.

In route planners, not SPARQL queries but *Earliest Arrival Time queries* (EAT) [9] need to be able to be solved. This is a question involving a time of departure, a departure stop and an arrival stop, expecting the earliest possible arrival time at the destination. More complex route planning questions exist,

---

[1] http://linkedconnections.org.
[2] https://developers.google.com/transit/gtfs/reference.
[3] Base URI: http://vocab.gtfs.org/terms#.
[4] http://linkeddatafragments.org.

such as the *Minimum Expected Arrival Time (*MEAT*)* [6] or *multi-criteria profile* queries [1,5,11]. The *Connection Scan Algorithm (*CSA*)* is an approach for planning that models the timetable data as a directed acyclic graph [6,9]. By topologically sorting the graph by departure time, the shortest path algorithm only needs to scan through connections within a limited time window to solve an EAT query. CSA can be extended to solving problems where it also keeps the number of transfers limited, as well as calculating routes with uncertainty [6]. The ideas behind CSA can scale up to large networks by using multi-overlay networks [9].

3 Linked Connections

In this section, we design our information system for public transport data, called the Linked Connections framework. As our datasets need to be interoperable with other data published on the Web, we choose HTTP as our *uniform interface.* We allow cross origin resource sharing and enable cache header to indicate when we expect a document to change. In order to document our identifiers using this uniform interface as well, RDF is chosen as a knowledge representation model. We further only define the hypermedia controls needed in each document as well as the vocabularies to be used.

To solve the EAT problem using CSA, timetable information within a certain time window needs to be retrievable. Other route planning questions need to select data within the same time window, and solving these is expected to have similar results[5] Instead of exposing an origin-destination API or a data dump, a Linked Connections server paginates the list of connections in departure time intervals and publishes these pages over HTTP. Each page contains a link to the next and previous one. In order to find the first page a route planning needs, the document of the entry point given to the client contains a hypermedia description on how to discover a certain departure time. Both hypermedia controls are expressed using the Hydra vocabulary[6].

The base entities that we need to describe are connections, which we documented using the LC Linked Data vocabulary[7]. Each connection entity – the smallest building block of time schedules – provides links to an arrival stop and a departure stop, and optionally to a trip. It contains two literals: a departure time and an arrival time. Linked GTFS can be used to extend a connection with public transit specific properties such as a headsign, drop off type or fare information. Furthermore, it also contains concepts to describe transfers and their minimum change time. For instance, when transferring from one railway platform to another, Linked GTFS can indicate that the minimum change time from one stop to another is a certain number of seconds.

---

[5] Also preprocessing algorithms [11] scan through an ordered list of connections to, for example, find transfer patterns [1].

[6] http://www.hydra-cg.com/spec/latest/core/.

[7] http://semweb.mmlab.be/linkedconnections#.

In the proof of concept built for this paper available at http://linkedcon nections.org, we implemented this hypermedia control as a redirect from the entry point to the page containing connections for the current time. Then, on every page, the description can be found of how to get to a page describing another time range. In order to limit the amount of possible documents, we only enable pages for each X[8] minutes, and do not allow pages describing overlapping time intervals. When a time interval is requested for which a page does not exist, the user-agent will be redirected to a page containing connections departing at the requested time. The same page also describes how to get to the next or previous page. This way, the client can be certain about which page to ask next, instead of constructing a new query for a new time interval.

A naive implementation of federated route planning on top of this interface is also provided, as a client can be configured with more than one entry point. The client then performs the same procedure multiple times in parallel. The connections streams are merge sorted as they are downloaded. A Linked Data solution is to ensure that the client knows how to link a stop from one agency to a stop from another.

4   Evaluation Design

We implemented different components in JavaScript for the Node.js platform. We chose JavaScript as it allows us to use both components both on a command-line environment as well as in the browser.

CSA.js A library that calculates a minimum spanning tree and a journey, given a stream of input connections and a query[9]

Server.js Publishes streams of connections in JSON-LD, using a MongoDB to retrieve the connections itself[10]

Client.js Downloads connections from a configurable set of servers and executes queries[11]

gtfs2lc A tool to convert existing timetables as open data to the Linked Connections vocabulary[12]

We set up 2 different servers, each connected to the same *MongoDB* database which stores all connections of the Belgian railway system for October 2015: 1. a *Linked Connections server* with an NGINX proxy cache in front, which adds caching headers configured to cache each resource for one minute[13] and compresses the body of the response using gzip; 2. a *route planning server* which

---

[8] X is a configurable amount.

[9] https://github.com/linkedconnections/csa.js.

[10] https://github.com/linkedconnections/server.js.

[11] https://github.com/linkedconnections/client.js.

[12] https://github.com/linkedconnections/gtfs2lc.

[13] The Belgian railway company estimates delays on its network each minute.

exposes an origin-destination route planning interface instead using the csa.js library[14].

These tools are combined into different set-ups:

Client-side route planning. The first experiment executes the query mixes by using the Linked Connections client. Client caching is disabled, making this simulate the LC *without cache* set-up, where every request could originate from an end-user's device.

Client-side route planning with client-side cache. The second experiment does the same as the first experiment, except that it uses a client side cache, and simulates the LC *with cache* set-up.

Server-side route planning. The third experiment launches the same queries against the full route planning server. The query server code is used which relies on the same CSA.js library as the client used in the previous two experiments.

In order to have an upper and lower bound of a real world scenario, the first approach assumes every request comes from a unique user-agent which cannot share a cache, and has caching disabled, while the second approach assumes one user-agent does all requests for all end-users and has caching enabled. Different real world caching opportunities – such as user agents for multiple end-users in a software as a service model, or shared peer to peer neighborhood caching [7] – will result in a scalability in-between these two scenarios.

We also need a good set of queries to test our three set-ups with. The iRail project[15] provides a route planning API to apps with over 100 k installations on Android and iPhone [3]. The API receives up to 400 k route planning queries per month. As the query logs of the iRail project are published as open data, we are able to create real query mixes from them with different loads. The first mix contains half the query load of iRail, during 15 min on a peak hour on the first of October. The mix is generated by taking the normal query load of iRail during the same 15 min, randomly ordering them, and taking the half of all lines. Our second mix is the normal query load of iRail, which we selected out of the query logs, and for each query, we calculated on which second after the benchmark script starts, the query should be executed. The third mix is double the load of the second mix, by taking the next 15 min, and subtracting 15 min from the requested departure time, and merging it with the second mix. The same approach can be applied with limited changes for the subsequent query mixes, which are taken from the next days' rush hours. Our last query mix is 16 times the query load of iRail on the 1st of October 2015. The resulting query mixes can be found at https://github.com/linkedconnections/benchmark-belgianrail.

These query mixes are then used to reenact a real-world query load for three different experiments on the three different architectures. We ran the experiments on a quad core Intel(R) Core(TM) i5-3340M CPU @ 2.70 GHz with 8 GB

---

[14] The route planning server uses the CSA.js library to expose a route planning API on top of data stored in a MongoDB. The code is available at https://github.com/linkedconnections/query-server.

[15] https://hello.irail.be.

of RAM. We launched the components in a single thread as our goal is to see how fast CPU usage increases when trying to answer more queries. The results will thus not reflect the full capacity of this machine, as in production, one would run the applications on different worker threads.

Our experiments can be reproduced using the code at https://github.com/ linkedconnections/benchmark-belgianrail. There are three metrics we gather with these scripts: the CPU time used of the server instance (HTTP caches excluded), the bandwidth used per connection, and the query execution time per LC connection. For the latter two, we use a per-connection result to remove the influence of route complexity, as the time complexity of our algorithm is $O(n)$ with n the total number of connections. We thus study the average bandwidth and query execution time needed to process one connection per route planning task.

5   Results

Figure 1 depicts the percentage of the time over 15 min the server thread was active on a CPU. The server CPU time was measured with the command pidstat from the sysstat package and it indicates the server load. The faster this load increases, the quicker extra CPUs would be needed to answer more queries.
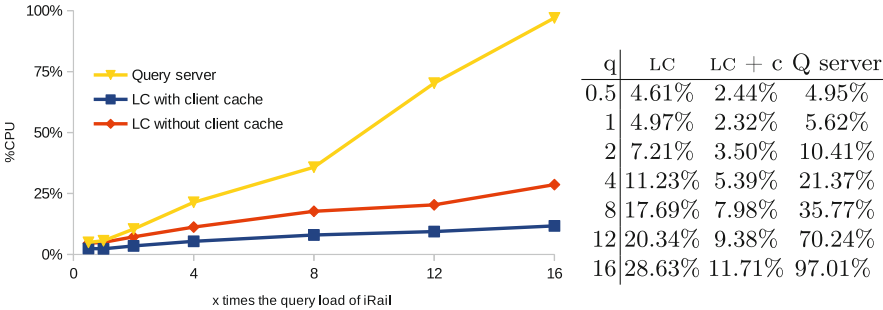

q	LC	LC + c	Q server
0.5	4.61%	2.44%	4.95%
1	4.97%	2.32%	5.62%
2	7.21%	3.50%	10.41%
4	11.23%	5.39%	21.37%
8	17.69%	7.98%	35.77%
12	20.34%	9.38%	70.24%
16	28.63%	11.71%	97.01%

Fig. 1. Server CPU usage under increasing query load shows that the Linked Connections server is more cost-efficient: more queries can be answered on one core.

When the query load is half of the real iRail load on October 1st 2015, we can see the lowest server load is the LC set-up with client cache. About double of the load is needed by the LC without client cache set-up, and even more is needed by the query server. When doubling the query load, we can notice the load lower slightly for LC with cache, and the other two raise slightly. Continuing this until 16 times the iRail query load, we can see that the load of the query server raises until almost 100%, while LC without cache raises until 30%, while with client cache, 12% is the measured server load.

In Fig. 2, we can see the query response time per connection of 90% of the queries. When the query load is half of the real iRail load on October 1st 2015,
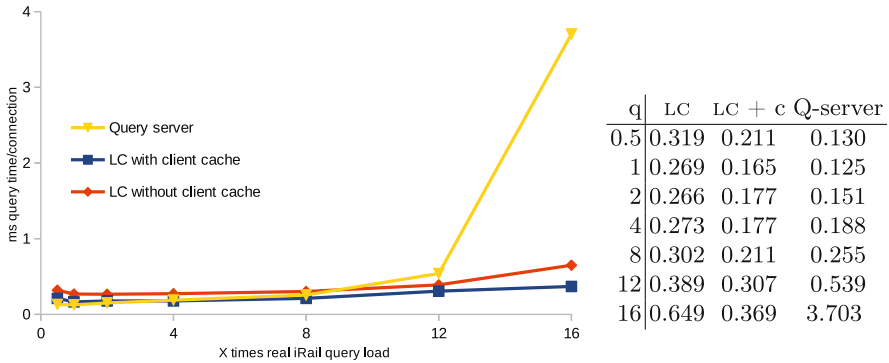
The figure contains a chart with y-axis labeled "ms query time/connection" ranging 0 to 4, x-axis labeled "X times real iRail query load" ranging 0 to 16, with legend:
- Query server
- LC with client cache
- LC without client cache

q	LC	LC + c	Q-server
0.5	0.319	0.211	0.130
1	0.269	0.165	0.125
2	0.266	0.177	0.151
4	0.273	0.177	0.188
8	0.302	0.211	0.255
12	0.389	0.307	0.539
16	0.649	0.369	3.703

Fig. 2. 90% of the journeys will be found with the given time in ms per connection under increasing query load.

we notice that the fastest solution is the query server, followed by the LC with cache. When doubling the query load, the average query execution time is lower in all cases, resulting in the same ranking. When doubling the query load once more, we see that LC with client cache is now the fastest solution. When doubling the query load 12 times, also the LC without client cache becomes faster. The trend continues until the query server takes remarkably longer to answer 90% of the queries than the Linked Connections solutions at 16 times the query load.

The average bandwidth consumption per connection in bytes shows the price of the decreased server load as the bandwidth consumption of the LC solutions are three orders of magnitude bigger: LC is 270B, LC with cache is 64B and query-server is 0.8B. The query server only gives one response per route planning question that is small. LC without client cache has a bandwidth that is three orders of magnitude bigger than the query server. The LC with a client cache has an average bandwidth consumption per connection that is remarkably lower. On the basis of these numbers we may conclude the average cache hit-rate is about 78%.

6   Conclusion

This paper introduced the Linked Connections framework for publishing queryable public transit data. We measured and compared the raise in query execution time and CPU usage between the traditional approach and Linked Connections. We could have hypothesized that we would find the origin-destination API approach to give faster response times, with a higher server CPU load that will increase even faster when the number of queries increase. We indeed achieved a better *cost-efficiency*: when the query-interface becomes saturated under an increasing query load, the lightweight LC interface only reached 1/4th of its capacity, meaning that the same load can be served with a smaller machine, or that a larger amount of queries can be solved using the same server. As the server load increases, the LC solution even give faster query results.

These result are strong arguments in favor of publishing timetable data in cacheable fragments instead of exposing origin-destination query interfaces when publishing data for maximum reuse is envisioned. The price of this decreased server load is however paid by the bandwidth that is needed, which is three orders of magnitude bigger. When route planning advice needs to be calculated while on for example a mobile phone network, network latency, which was not taken into account during the tests, may become a problem when the cache of the device is empty. An application's server can however be configured with a private origin-destination API, which in its turn is a consumer of a Linked Connections dataset, taking the best from both worlds.

Our goal was to enable a more flexible public transport route planning ecosystem. While even personalized routing is now possible, we also lowered the cost of hosting the data, and enabled in-browser scripts to execute the public transit routing algorithm. Furthermore, the query execution times of queries solved by the Linked Connections framework are competitive. Until now, public transit route planning was a specialized domain where all processing happened in memory on one machine. We hope that this is a start for a new ecosystem of public transit route planners.

References

1. Bast, H., Carlsson, E., Eigenwillig, A., Geisberger, R., Harrelson, C., Raychev, V., Viger, F.: Fast routing in very large public transportation networks using transfer patterns. In: Berg, M., Meyer, U. (eds.) ESA 2010. LNCS, vol. 6346, pp. 290–301. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15775-2_25
2. Colpaert, P., Ballieu, S., Verborgh, R., Mannens, E.: The impact of an extra feature on the scalability of linked connections. In: COLD Workshop at ISWC2016 (2016)
3. Colpaert, P., Chua, A., Verborgh, R., Mannens, E., Van de Walle, R., Vande Moere, A.: What public transit API logs tell us about travel flows. In: WWW 2016: 25th International World Wide Web Conference (2016)
4. Colpaert, P., Llaves, A., Verborgh, R., Corcho, O., Mannens, E., Van de Walle, R.: Intermodal public transit routing using linked connections. In: Proceedings of the 14th International Semantic Web Conference: Posters and Demos, October 2015
5. Delling, D., Pajor, T., Werneck, R.F.F.: Round-based public transit routing. In: Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX 2012) (2012)
6. Dibbelt, J., Pajor, T., Strasser, B., Wagner, D.: Intriguingly simple and fast transit routing. In: Bonifaci, V., Demetrescu, C., Marchetti-Spaccamela, A. (eds.) SEA 2013. LNCS, vol. 7933, pp. 43–54. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38527-8_6
7. Folz, P., Skaf-Molli, H., Molli, P.: CyCLaDEs: a decentralized cache for triple pattern fragments. In: Sack, H., Blomqvist, E., d'Aquin, M., Ghidini, C., Ponzetto, S.P., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9678, pp. 455–469. Springer, Cham (2016). doi:10.1007/978-3-319-34129-3_28
8. Quercia, D., Schifanella, R., Aiello, L.M.: The shortest path to happiness: recommending beautiful, quiet, and happy routes in the city. In: Proceedings of the 25th ACM Conference on Hypertext and social Media, pp. 116–125. ACM (2014)

9. Strasser, B., Wagner, D.: Connection scan accelerated. In: Proceedings of the Meeting on Algorithm Engineering & Expermiments, pp. 125–137. Society for Industrial and Applied Mathematics (2014)
10. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple pattern fragments: a low-cost knowledge graph interface for the Web. J. Web Semant. **37**, 184–206 (2016)
11. Witt, S.: Trip-based public transit routing. In: Bansal, N., Finocchi, I. (eds.) ESA 2015. LNCS, vol. 9294, pp. 1025–1036. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48350-3_85