Compressed Data Structures for Trajectory Representation

Autor: Daniil Galaktionov Hodovaniuk

Tesis doctoral UDC / 2020

Directores:
Antonio Fariña Martínez
Nieves Rodríguez Brisaboa

Tutor y Director por parte de la empresa:
Eduardo Rodríguez López

Compressed Data Structures for Trajectory Representation

Autor: Daniil Galaktionov Hodovaniuk

Tesis doctoral UDC / 2020

Directores:
Antonio Fariña Martínez
Nieves Rodríguez Brisaboa

Tutor y Director por parte de la empresa:
Eduardo Rodríguez López

Programa Oficial de Doutoramento en Computación

PhD thesis supervised by
Tesis doctoral dirigida por

Antonio Fariña Martínez
Departamento de Computación
Facultad de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981 167000 ext. 1352
Fax: +34 981 167160
antonio.farina@udc.es

Nieves Rodríguez Brisaboa
Departamento de Computación
Facultad de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981 167000 ext. 1243
Fax: +34 981 167160
brisaboa@udc.es

Tutor y director responsable por parte de la empresa

Eduardo Rodríguez López
Enxenio S.L.
Calle José Luis Bugallal Marchesi
№ 20 1 Izq., 15008 A Coruña (España)
Tel: +34 981 913 768
edu@enxenio.es

Antonio Fariña, Nieves Rodríguez Brisaboa y Eduardo Rodríguez, como directores, acreditamos que esta tesis cumple los requisitos para optar a los título de doctor industrial e internacional, y autorizamos su depósito y defensa por parte de Daniil Galaktionov Hodovaniuk cuya firma también se incluye.

In the memory of my grandmother.

Acknowledgements

I had to admit to myself that I was stalling before getting to write this page at last. No wonder that some say this is the hardest part! It is stressful just to think of the responsibility: so many of you helped and supported me along the way, what if I were to forget mentioning someone? Indeed, any mistakes in these few paragraphs will be the hardest to live with.

It seems appropriate to start with whoever was closest to oneself during these last years. In some cases, that would be their thesis advisor, but (fortunately) not in my case. Camilia, my beautiful and caring wife would have not tolerated such thing! I would have not met her if I had not traveled down the path that culminated in this thesis, which happened in part thanks to her support, for which I am grateful. And of course I would also like thank my parents, who kept in touch month after month, and encouraged me to follow through with whatever path I would choose.

Next, I must thank everyone from my lab and Enxenio who have made this possible. Among them, in chronological order: First, Nieves, who made me an offer that I could not refuse, and looked after me until the end. Second, Guillermo and Luaces, who in their infinite patience, so characteristic of a PhD, would take the effort to train my younger self. Next came Fari, who mentored me and performed a great role in cultivating the required patience in me. Even if for a brief time, Andrea has also played a major role both in this research and in my education. Also a huge thanks to everyone who advised and helped me along the way: Fernando, Nelly, Adrián, Suilén, Tirso, and Jeremy, in no particular order.

Finally, I would like to thank my friends Sara, Luis, and Chon, who stood by my side all the way more times than I could ask for. This is for all of you!

This thesis has received funding from MICIU/Ayudas para contratos para la formación de doctores en empresas (Doctorados industriales): DI-15-07576; MICIU/FEDER-UE BIZDEVOPSGLOBAL: RTI-2018-098309-B-C32; Xunta de Galicia/FEDER-UE CSI: ED431G/01; GRC: ED431C 2017/58; GEMA: IN852A 2018/14; Xunta de Galicia/GAIN Innovapeme: IN848D-2017-2350417; MINECO-AEI/FEDER-UE Datos 4.0: TIN2016-78011-C4-1-R; ETOME-RDFD3: TIN2015-69951-R; and EU H2020 MSCA RISE BIRDS: 690941.

Agradecimientos

Debo admitir que me encontraba estancado antes de ocuparme al fin de esta página. ¡No es de extrañar que se diga que esta es la parte más difícil! Sólo pensar en la responsabilidad que conlleva resulta estresante: tantos de vosotros me han ayudado y apoyado por el camino, ¿y si me olvido de mencionar a alguno? Desde luego, cualquier error en estos párrafos serán los más dolorosos de recordar.

Parece apropiado comenzar con quien haya estado más cerca de uno durante estos últimos años. Para algunos, esta persona es su tutor de tesis, aunque (afortunadamente) no ha sido así en el mío. ¡Camilia, mi hermosa esposa, no habría tolerado tal cosa! No la habría conocido de no haber recorrido el camino que culminó en esta tesis, y le estaré siempre agradecido porque su apoyo logró que finalmente esta tesis se llegase a realizar. Por supuesto, también me gustaría agradecer a mis padres, quienes se mantuvieron en contacto mes tras mes, y me animaron siempre a continuar con cualquier camino que hubiera elegido.

Después de esto, me gustaría agradecerles a todos mis compañeros de laboratorio y de Enxenio que han hecho esto posible. Entre ellos, por orden cronológico: Primero Nieves, quien me hizo una oferta que no pude rechazar, y se ocupó en cuidar de mí hasta el final. En segundo lugar, Guillermo y Luaces, quienes en su infinita paciencia, tan característica en un Doctor, habían asumido el esfuerzo de formar a mi versión más joven. Después vino Fari, quien me guió, desempeñando un gran papel en cultivar la paciencia requerida en mí. Incluso si fue durante un breve tiempo, Andrea también jugó un rol importante tanto en esta investigación como en mi educación. También quiero darles las gracias a todos los que me han aconsejado y ayudado a lo largo del camino, sin ningún orden en especial: Fernando, Nelly, Adrián, Suilén, Tirso y Jeremy.

Finalmente, me gustaría agradecer a mis amigos Sara, Luis y Chon, quienes se mantuvieron a mi lado todo este tiempo más veces de las que podría pedir. ¡Esto es para todos vosotros!

Esta tesis ha recibido fondos de MICIU/Ayudas para contratos para la formación de doctores en empresas (Doctorados industriales): DI-15-07576; MICIU/FEDER-UE BIZDEVOPSGLOBAL: RTI-2018-098309-B-C32; Xunta de Galicia/FEDER-UE CSI: ED431G/01; GRC: ED431C 2017/58; GEMA: IN852A 2018/14; Xunta de Galicia/GAIN Innovapeme: IN848D-2017-2350417; MINECO-AEI/FEDER-UE Datos 4.0: TIN2016-78011-C4-1-R; ETOME-RDFD3: TIN2015-69951-R; y EU H2020 MSCA RISE BIRDS: 690941.

Abstract

The proliferation of GPS devices in smartphones, vehicles and sport wearables on the one hand, and geolocation mechanisms (such as smart cards in public transportation) on the other hand, have led to an unprecedented ability to gather and store trajectories that originate from people's movements during their daily schedules. However, no standard data models exist to represent these trajectories and, in addition, neither traditional databases nor new *NoSQL* databases are adequate for the representation and exploitation of the complex spatio-temporal data that make up such trajectories. This general outlook is even more complex once we consider that whenever we are storing information related to the context of public transportation passengers, customers inside a mall, or simply vehicles moving in a city we must deal with a true Big Data scenario in which guaranteeing an efficient response can be very challenging.

Consequently, in this thesis we address the design of compact data structures for the representation of the followed trajectories, both in the context of vehicles and/or people moving in urban or periurban spaces, as in the context of itineraries of commuters in public transportation. Apart from designing these compact data structures that allow us to represent the Big Data scenario usually seen in this application domain, we have also designed the algorithms that allow the efficient exploitation of the underlying information.

We have implemented algorithms that not only to solve the classic spatio-temporal queries, such as obtaining the position of a moving object at a time instant, reconstructing the trajectory of an object, or even spatio-temporal window queries (which objects are inside a spatial range either within a time window or at a time instant), but also solve more specialized queries for the analysis of the trajectories that travelers make. For instance, we have designed algorithms to query the number of travelers that start (or finish) their trip in a certain place within a given time interval, or the number of travelers that switch from one line from the public transportation network to another one using a particular stop, or even the number of travelers that had started their trip in a certain place (which can be either a stop or a whole neighborhood) and finished it in another place.

Both the designed structures and the querying algorithms, which are available at `https://github.com/dgalaktionov/compact-trip-representation`, have been experimentally evaluated. With these structures we were able to represent, in a compact space of 100 MiB, a collection of approximately a million and a half of taxi trajectories, or alternatively ten million trajectories consisting of itineraries over public transportation networks (the latter being more compressible). In both cases, we can solve most of the considered exploitation queries in the order of microseconds, with algorithms that scale logarithmically with respect

to the increase in the number of stored trajectories.

Finally, considering that this work is considered an industrial thesis, and that this requires showing that the research performed is of clearly applied nature, we have developed a web application with Geograhic Information Systems technology, which integrates with our compressed structures and algorithms instead of relying on common spatial databases. This application, which provides a simple and intuitive user interface that represents the map of a transportation network, enabled an end user to run the aforementioned querying algorithms over a large collection of historic trajectories. Likewise, this interface presents the query results in a graphical and intuitive way.

Resumen

La proliferación de por un lado de dispositivos GPS en smartphones, vehículos o pulseras de deporte, y por otro, de otros mecanismos de geolocalización (como las tarjetas de pago de trasporte público), han dado lugar a una capacidad inédita de obtener y almacenar las trayectorias que generan las personas al moverse durante sus quehaceres diarios. Sin embargo, no existen modelos de datos estándar para representar dichas trayectorias, además de que ni las bases de datos tradicionales, ni las nuevas bases de datos *NoSQL* se adecúan bien a la representación y explotación de esos datos complejos de naturaleza espacio-temporal que son las trayectorias. Para hacer más complejo aún el panorama, se constata además que cuando se quieren almacenar trayectorias de viajeros de transporte público, o de clientes en centros comerciales, o simplemente de personas o vehículos moviéndose por una ciudad hay que enfrentarse a un verdadero escenario Big Data en el que la eficiencia en la respuesta a las consultas se hace muy difícil.

Por todo ello, en esta tesis se aborda el diseño de estructuras de datos compactas para la representación de las trayectorias seguidas, por un lado, por vehículos y/o personas que se mueven por las calles de un entorno urbano o periurbano acotado, y por otro los itinerarios de viajeros de transporte público. Además de diseñar esas estructuras de datos compactas, que permiten representar ese escenario Big Data habitual en estos dominios de aplicación, se han diseñado los algoritmos que permiten la explotación eficiente de dichos datos.

Hemos implementado algoritmos que, además de resolver las consultas espacio-temporales clásicas, tanto las de posición de un objeto en un tiempo, o trayectoria de un objeto durante un intervalo temporal, como las consultas de rango espacio-temporal (qué objetos están en una ventana del espacio en un instante o intervalo temporal) resuelven también consultas más especializadas para el análisis de trayectorias de viajeros. Por ejemplo, hemos diseñado algoritmos para consultar el número de viajeros que inician (o terminan) su viaje en un lugar dado dentro de un cierto intervalo temporal, o el número de viajeros que conmutan de una línea a otra de la red de transporte público en una parada concreta, o incluso el número de viajeros que inicia su viaje en cierto lugar (parada o barrio) y lo termina en otra parada o barrio determinados.

Tanto las estructuras de datos diseñadas como todos los algoritmos de consulta, que están disponibles en `https://github.com/dgalaktionov/compact-trip-representation`, han sido evaluados experimentalmente. Con estas estructuras es posible representar en un espacio de 100 MiB una colección de aproximadamente un millón y medio de trayectorias de taxis, o alternativamente diez millones de trayectorias consistentes de itinerarios sobre redes

de transporte público, al ser éstas últimas más compactas. En ambos casos, podemos resolver la mayor parte de las consultas de explotación planteadas en el orden de microsegundos, con algoritmos que escalan de forma logarítmica con respecto al incremento en el número de trayectorias almacenadas.

Por último, considerando que este trabajo está considerado como una tesis industrial, lo cual requiere demostrar que el trabajo investigador realizado es de naturaleza aplicada, hemos desarrollado una aplicación web con tecnología de Sistemas de Información Geográfica que, en vez de trabajar sobre una base de datos espacial convencional, utiliza la estructura comprimida y los algoritmos para su explotación diseñados en la tesis. Esa aplicación facilita, mediante una sencilla e intuitiva interfaz de usuario que representa el mapa de la red de transporte, el lanzamiento de los algoritmos diseñados sobre un amplio conjunto de trayectorias de viajeros. Del mismo modo esa interfaz presenta los resultados de las consultas de modo gráfico e intuitivo.

Resumo

A proliferación de por un lado dos dispositivos GPS en smartphones, vehículos ou brazaletes deportivos e por outra banda dos mecanismos de xeolocalización (como as tarxetas de pago do transporte público), teñen dado lugar a unha capacidade sen precedentes para obter e almacenar as traxectorias que a xente xera ao moverse durante as súas tarefas diarias. Sen embargo, non hai modelos de datos estándar para representar ditas traxectorias, e ademais de que nin as bases de datos tradicionais nin as novas bases de datos *NoSQL* se adecúan ben á representación e explotación dos datos tan complexos e de natureza espazo-temporal que son as traxectorias. Para complicar aínda máis o panorama, tamén se comproba que cando se queren almacenar traxectorias de viaxeiros de transporte público, ou de clientes en centros comerciais, ou simplemente de persoas ou vehículos que se desprazan por unha cidade, se ten que afrontar un verdadeiro escenario de Big Data no que a eficiencia na resposta ás consultas se fai moi difícil.

Por iso, esta tese trata do deseño de estruturas compactas de datos para a representación dos camiños seguidos, por un lado, por vehículos e/ou persoas que se desprazan polas rúas dun contorno urbano ou periurbano delimitado, e por outro lado os itinerarios de viaxeiros en transporte público. Ademais de deseñar estas estruturas compactas de datos, que permiten representar dito escenario Big Data habitual nestes dominios de aplicación, deseñáronse algoritmos que permiten a explotación eficiente dos devanditos datos.

Estes algoritmos, ademais de resolver as clásicas consultas espazo-temporais, tanto a posición dun obxecto nun instante dado, como a traxectoria dun obxecto durante un intervalo de tempo, así como as consultas de rango espazo-temporal (que obxectos están nun rango do espazo nun intre ou nun intervalo temporal) tamén permiten resolver consultas máis especializadas para a análise de traxectorias de viaxeiros. Por exemplo, deseñamos algoritmos para comprobar o número de viaxeiros que inician (ou rematan) a súa viaxe nun determinado lugar nun certo intervalo de tempo, ou o número de viaxeiros que cambian dunha liña a outra da rede de transporte público nunha parada concreta, ou incluso o número de viaxeiros que comezan a súa viaxe nun determinado lugar (parada ou barrio) e rematan noutra parada ou barrio específico.

Tanto as estruturas de datos deseñadas como todos os algoritmos de consulta, dispoñibles en `https://github.com/dgalaktionov/compact-trip-representation`, foron avaliados experimentalmente. Con estas estruturas é posible representar nun espazo de 100 MiB unha colección de aproximadamente un millón e medio de traxectos de taxi ou, alternativamente, dez millóns de traxectos consistentes en itinerarios en redes de transporte público, por ser estes últimos máis compactos. Nos dous casos, podemos resolver a maioría das

consultas de explotación plantexadas na orde de microsegundos, con algoritmos que escalan logarítmicamente con respecto ao aumento do número de traxectorias almacenadas.

Finalmente, dado o carácter de tese industrial deste traballo, foi necesario que a investigación realizada tivese un carácter claramente aplicado, polo que se implementou unha aplicación web con tecnoloxía de Sistemas de Información Xeográfica que, no canto de traballar nunha base de datos espacial convencional, usa a estrutura comprimida e algoritmos de explotación deseñados na tese. Esta aplicación facilita, mediante unha interface de usuario sinxela e intuitiva que representa o mapa da rede de transporte, o lanzamento dos algoritmos deseñados nun amplo conxunto de rutas de pasaxeiros. Do mesmo xeito, dita interface presenta os resultados das consultas dun xeito gráfico e intuitivo.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

AFC	Automated Fare Collection. 2, 95, 114
CRS	Coordinate Reference System. 82, 83, 86, 88
CSA	Compressed Suffix Array. xxi, xxiii, 22, 23, 25–27, 29–32, 34, 38, 43, 44, 46, 50, 51, 55, 58, 60–63, 66–68, 70, 71, 73–77, 109, 117, 118
CTR	Compact Trip Representation. xxi, xxiii, 5, 25, 26, 28–32, 35, 37, 38, 40, 41, 43–46, 49, 51, 55, 56, 58, 62, 66, 109, 110, 117, 118
DBMS	Database Management System. 94
GIS	Geographic Information System. 5, 6, 81, 82, 84, 87, 89, 90, 93, 109, 110, 117, 118
GTFS	General Transit Feed Specification. xxii, 4, 41, 89, 90, 94, 115
HTWT	Hu-Tucker Wavelet Tree. xxi–xxiii, 20, 29, 32–34, 36, 38, 41, 43–45, 47–54, 117
RDBMS	Relational Database Management System. 96
SAT	Summed Area Table. xxi, 15, 16, 55, 64, 110, 118
T-Matrices	Trip-Matrices. xxii, xxiii, 55, 58, 59, 64, 65, 70–73, 76–78, 110, 118
TMS	Tile Map Service. 88, 91, 99, 100
TTCTR	Topology&Trip-aware Compact Trip Representation. xxii, xxiii, 5, 55, 58–64, 66, 70–78, 110, 117, 118

UTM	Universal Transverse Mercator. 83
WFS	Web Feature Service. 89, 91
WM	Wavelet Matrix. xxi–xxiii, 21, 29, 32–34, 36, 38, 41, 43–45, 47–55, 58, 62, 63, 66, 70, 71, 73–77, 117
WMS	Web Map Service. 88
WT	Wavelet Tree. xxi, 15, 18–22, 25–27, 29, 32–34, 37, 41, 109, 117, 118
XCTR	eXtended Compact Trip Representation. xxii, xxiii, xxv, 5, 55, 58, 59, 62–64, 66–78, 93–98, 100, 101, 110, 117, 118

Chapter 1

Introduction

1.1 Motivation

The last years have seen a widespread adoption of technological methods focused in registering the movements of individuals, most notably the *smartphone* apps for navigation and map visualization, which often collect the followed GPS trajectories. Similar applications have also been adopted by companies that manage fleets of vehicles, such as taxi and emergency services. When the data about a large amount of these individual trajectories is collected and aggregated, it can be used to infer mobility patterns [LKG+12] or, if the collection is complete enough, to model a traffic scenario from the collected sample, as for example was shown in [JL09] with the streets of Hong Kong.

Moreover, in the context of public transportation networks, transportation companies have adopted numerous advances in wireless technologies, sensor networks (especially those related to RFID) and ubiquitous computing, leading to a widespread adoption of passenger tracking technology by public transportation services, making the collection of large amounts of data about the travel habits of these passengers[1] easier than ever before. This in turn has opened the door for the exploitation of this kind of information to study the demand (usage) of a network, as opposed to the well-known techniques to analyze the offer (routes, timetables, etc...). Examples of useful analysis tasks can be the measures of accessibility and centrality indicators, referred to how easy is to reach different locations or how important certain stops are within a network [MTA07, EGL11, WZTL15]. All these measures are based on some kind of counting queries that determine the number of *distinct* trips that occur within a spatial and/or temporal window.

To enable these new kinds of *demand* studies, it is imperative to develop mechanisms to efficiently persist and manage these vast (and always increasing) collections of data. When we also take into account that efficient query patterns need to be supported for this data to be "useful", the solution clearly constitutes an emerging technological challenge that is being approached from several different domains, and hundreds of ad-hoc solutions have been implemented by all the *Smart Cities* around the globe.

[1] Alternatively called "users" in the context of transportation companies.

Therefore, it follows that a practical representation that supports efficient indexing for not only collected GPS trajectories, but also collections of trips over a transportation network, would have numerous possible applications.

In [TCY+18] we can see how it is possible to combine GPS trajectories with Automated Fare Collection (AFC) data to recreate complete trips and study the ridership by area. Alternatively, in [WLS+18] the complete trips are inferred from the AFC data, to later analyze behaviour patterns and preferences of the travelers with the goal of improving the efficiency of the network. Another application that is enabled by such analysis is targeted advertising [ZGN+17], as the interests of a user can be profiled by their travel patterns. Other works focus on analyzing the usage of individual stops or stations, such as [CSC12], where the authors determine that congestion times in the metro network of London are predictable and occur in narrow time intervals. Armed with such information, an user may choose a different travel pattern to avoid the crowd and enhance their overall experience.

When we consider practical studies focused on trajectories over street networks, we can find works centered around studying taxi ridership. One notable example is [YZZX13], that discusses a two-way taxi recommendation system, where taxi drivers are pointed to the most profitable parking spaces while passengers are directed to the street segments with a high probability of finding a vacant taxi.

One key observation from all the works referenced above is that a mere collection of trajectories or time-stamped points over a two-dimensional space of latitude and longitude would not be rich enough to perform these studies. They are therefore required to work with a representation that allows for some degree of *semantic* information. At the very least, that information must include references to network elements (stops, lines or streets), and sometimes even some (anonymized) user identifier. Therefore, we require a representation that differs from the traditional spatial indexes and databases, as it must support efficient access methods based on network elements.

1.2 Problem definition

Considering the nature of the discussed problems, and also the different sources of information, we have identified two distinguishable contexts for transportation analysis:

1.2.1 Trajectories over urban streets

In this context, a trajectory can start anywhere, at any time, and follow any arbitrary path of segments along of streets, which can have either of two alternative graph representations: assign a vertex for each intersection and edges for street segments that connect these intersections, or alternatively, assign a vertex for each street segment with no intersections and edges that connect navigable street segments. In any case, the geolocated trajectory must be mapped to a path of street segments in order to enable analyzing the movement patterns between locations (which are always bound to streets in urban contexts) and average traffic load at a given time of the day. Trajectories of taxis, bicycles, or vehicle fleets fall into this context. For these systems, queries of interest may involve retrieving the points of interest where these trajectories could end, or street segments that could be part of a given path.

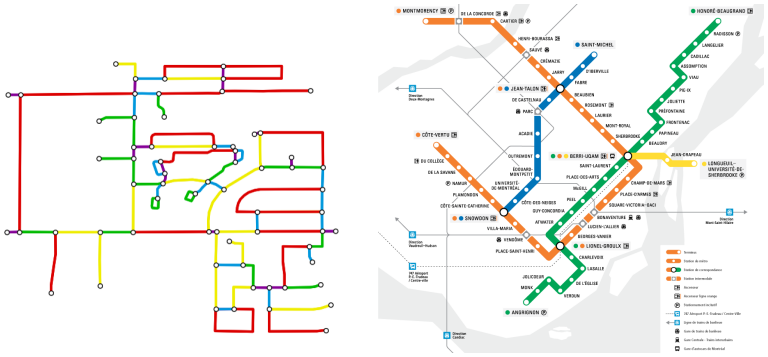


Figure 1.1: Example of a typical urban street layout (left) and a subway network (right).

Sources: [Boe17] (left), `http://www.stm.info/fr/infos/reseaux/metro` (right)

Our definition also requires to be able to speak for a concept of time. For these trajectories where an object moves freely over the network of streets, a representation of the time intervals when each street segment was traversed may be considered. In order to achieve a compact representation, time would be expressed in discrete intervals ranging from one to thirty minutes. Larger sizes for these time intervals may not be practical since many of the trajectories would be completed in less than thirty minutes, thus fitting completely in the same interval, thus defeating the purpose of representing a time interval for every segment of the path.

1.2.2 Trips over public transportation networks

In this case, instead of individual trajectories we consider **trips**, which must start at predefined points (stops or stations) at set times that are defined by the vehicles that stop at those points. These vehicles follow predefined paths along these points, following lines. Therefore, a trip follows a path along a network of stops and lines, which can be formed over a street network as in the case of buses, or hold very little relation to the streets, as in the case of a subway. However, even when the transportation network is related to a street network, it is more interesting to define the trips using the stops and lines from the transportation network, as enumerating the individual street segments that are traversed would have a high redundancy with no clear benefit. Note that defining user trips over these network elements will still produce redundant collections, as users that travel in the same vehicle would produce identical parts of trips. This property is one of the keys to achieve a compact representation, since the analysis tasks revolve around the usage and trip patterns of network elements (stops and lines) instead of individual trips. This context applies to bus and metro systems, along train lines and periurban buses.

In this working context, we will operate with a *network model*. For a public

transportation network, it could be pertinent to consider a more rich representation than a mere graph of stops and lines, thus also taking into account the routes formed by transportation vehicles, such as buses or trains, visiting stops at set times and allowing commuters to board or alight at them. A well-known model that includes these network elements, among others less interesting for our problem, is the General Transit Feed Specification (GTFS)[2], which is widely adopted by open data platforms in numerous *Smart Cities*.

Therefore, a *trip* will be defined as a path formed by a sequence of stops, that was traversed by a single passenger/commuter in one trip, with an origin and a final destination. In this definition, we must consider some practical limitations to the nature of a trip, as one could argue whether commuters that take more than one hour to switch a line are actually switching or just finalized their initial trip and are starting a second one with some new destination. These cases are complicated to unambiguously decide in practice, and therefore our approach will tend to set hard limits on waiting times and walking distances between stops for a single trip.

In a network model where the routes are formed by transport vehicles that follow lines, there would be no need for representing the exact time at which each user has boarded on a stop. We will only have to asign a route identifier, as the stopping times would be already available in our modelled network, thus avoiding some redundancy in the representation of trips.

Massive data collection techniques exist for both of the contexts discussed above, as will be later seen in Section 2.1, leading to the problem of efficiently handling these vasts amounts of information that both contexts produce. Apart from the usual well-known *Big Data* solutions (Hadoop, Spark, Druid...), there is an ongoing research line on the application of succint data structures for some of these goals. In particular, it is possible to apply many of the techniques from the field of *Compact Data Structures* to build autoindexed representations that support efficient query patterns tailored for specific information needs, while offering some sort of compression with respect to a more traditional representation.

A usable solution would also require an user interface that enables the exploitation of this information by researchers, transportation companies, city administrations and any other kind of end users. This interface must, at the very least, allow visualizing the network elements on a map, also granting the ability to make queries over these elements in an intuitive and responsive way, while respecting the usual quality principles of any user-oriented software of this kind.

1.3 Contributions

As the two contexts from Section 1.2 lead to different ways of structuring and querying the information, it is natural to expect at least two different representations, one for each context. For this reason, in this work we have applied compact data structures and techniques to design novel representations that are able to handle massive collections of data related to user moving and transportation habits. Our proposed representations cover both contexts, as well as offer a fair trade-off for different query needs, while at the same time ensuring that our proposals can be implemented in a real-world scenario, for which

[2] `https://developers.google.com/transit/gtfs/`

we evaluate the performance of our representations with realistic query cases over real datasets.[3]

Furthermore, as a proof of the practicability of our approach, we have developed an end-user application that is based on our representations, and allows a user to perform queries through a Geographic Information System (GIS) web interface. Unlike traditional GIS interfaces, ours is focused on offering convenient methods to query historical user trips by the network elements instead of purely spatial relationships, and achieves far superior performance when compared to systems backed by traditional database systems.

In conclusion, we present an end-to-end platform around representations based on compact data structures to process, store, query and visualize mined public transportation usage data. To the best of our knowledge, this is the first work to accomplish building such integrated platform, although other works exist that contemplate the use of compact data structures for trajectories or moving objects (see Section 2.3).

1.4 Outline

The rest of this thesis is structured as follows: In the following Chapter 2, we review the literature on existing methods for trajectory extraction and indexing. The former is interesting to our work as it studies different approaches to obtain the trips over public transportation systems, which our work focuses on, while the latter discusses alternatives for supporting some of the traditional queries over trajectories, that are often based on secondary memory.

After that, our contributions are grouped into two parts:

- In **Part I**, we propose several representations based on compact data structures to efficiently handle the analysis of trips for both transportation contexts discussed in Section 1.2. It is divided in three chapters:
  - In Chapter 3, we describe the underlying structures that are used by our representations, with a brief description of the memory requirements of each one of them, as well as the temporal complexities of their main algorithms.
  - Chapter 4 is devoted to Compact Trip Representation (CTR), our representation for trajectories in the context of urban streets. We use separate structures for the spatial and temporal representations, and combine them so that they can be used to solve spatio-temporal queries.
  - In Chapter 5, we discuss the problems that are specific for trips over public transportation networks, and provide two alternative representations, Topology&Trip-aware Compact Trip Representation (TTCTR) and eXtended Compact Trip Representation (XCTR), based on a common model, along with an additional complementary structure that can be used to accelerate some of the aggregation queries.
- In **Part II**, we present an interface designed to aid on the decision making process of a public transportation company, that relies on the representations from Part I. This part consists of two chapters:

[3] When needed, the real data was augmented or mixed with synthetic information.

- Chapter 6 introduces the reader to some of the basic concepts of GIS, as well as the technologies used in our developed interface.
- Chapter 7 contains the detailed description of our application, discussing its architecture and our user interface to analyze historical information of user trips over public transportation networks.

After these parts, **Part III** summarizes our contributions in one single concluding Chapter 8, where we also discuss the future developments planned for the work exposed in this thesis.

Finally, we also include two appendices: Appendix A enumerates the relevant research works that have derived from this thesis, while Appendix B provides a summary of the overall thesis in the Spanish language, as required by the current regulations in the PhD program that this thesis is submitted for.

Chapter 2

State of the art in trajectory extraction and representation

This chapter is devoted to provide a context to our contributions with a literature review of tangential works. We start by discussing methods of collecting useful trajectory data, followed by a review of mobile objects and models for the trajectories they generate, as well as types of queries that are often found in mobile objects research. Finally, we look into the most relevant works in trajectory indexing, which contrast with our contributions as the latter are based on auto-indexed representations, and are focused on problems specific to the study of transportation demand and travel patterns on transportation networks.

2.1 Trajectory extraction

In order to make significant conclusions about travel patterns, it is essential to be able to collect a large enough collection of trips so that it would become representative of the overall usage within a time span. For this problem, crowd-sourcing can be a viable approach to record the trajectories of public transportation users, as done in [ZTG+11], where the users were rewarded with real information on the bus location, estimation of arrival time and fullness in exchange of recording the GPS trace on their own bus trips.

Currently there are several known techniques that would allow to collect data regarding users' trips over a public transportation network. Numerous works exist where those trajectories are mined from the transactions of smart cards [BC+15, WLL14]. This can be complemented with information derived from GPS devices, as shown in [MW14]. Alternatively, reliable trajectories may be extracted relying on positioning obtained from cellular networks, as proven by [LWW+17].

Because smart card methods usually provide only information about boarding stops, there are works that study the challenge of inferring alighting stops from passengers [Wan11]. In addition, the authors of [TCMBR14] have specifically tackled the challenge of reconstructing full trajectories, accounting for trip-chaining, by using data obtained from smart cards. Furthermore, in [AAMF16] it is also proven that not only the alighting stops,

but also the (last) destination stop of a trip can be estimated from boarding data gathered by a smart card, within a reasonable accuracy. This is possible because a transportation network user typically makes a return trip at another time of the day, as happens often with people that commute from their homes to work and back. We find these methods particularly interesting, as a significant portion of the query patterns we propose in order to analyze passenger movements rely on knowing about line switches (trip-chaining) and the ultimate destination of a trajectory.

However, there is little research on methods for exploitation of this extracted information, in order to analyze and improve the efficiency of a transportation network, which constitutes the scope of our work. This summarized review proves that, although we were not able to access real data from a public transportation company for this work, such curated information about users' trips can indeed be obtained and used for our proposed representations.

2.2 Models of trajectory and types of queries

A good place to begin searching for practical models of trajectory representation is the vast amount of work on designing data models for moving objects [WXCJ98, SWCD97, GBE+00, GBE+03, Spa01, FGNS00, EGSV99, GS05]. Basically, a data model for moving objects represents the continuous change of the location of an object over time, this way defining a trajectory.

Handling moving objects can be seen as a big data problem, where solutions may typically differ in the representation of location, contextual or environmental information where the movement takes place, the time dimension (which can be continuous or discrete) and the level of abstraction or granularity on which the trajectories are described [DIGV15]. A common classification of trajectories distinguishes free from network-based trajectories. *Free trajectories* or Euclidean trajectories are a sequence of GPS points represented by an ad-hoc data type of moving points [WXCJ98, SWCD97, GBE+00]. *Network-constrained* trajectories are a temporal-ordered sequence of locations on networks. This trajectory model includes a data type for representing networks and for representing the relative location of static and moving points on the network [GdAD06]. When we want to represent trajectories over an urban street network, it is often useful to deal with *network-matched trajectories*, as this will allow to represent large collections of trajectories more effectively. The process of obtaining these network-matched trajectories from GPS points is called *map-matching* [BPSW05]. This process can also be done online, as recently shown in [DYGL15], where all the processing is done in the server, eliminating the need for any map or network representation at the moving object side.

The definition of trajectories at an abstract level must be materialized in an internal representation with access methods for query processing. An early and broad classification of spatio-temporal queries for *historical positions* of moving objects [PJT00] identifies coordinate- and trajectory-based queries. Coordinate-based queries include the well-known *time-slice*, *time-interval* and *nearest-neighbor queries*. Typical examples are *"find objects or trajectories in a region at a particular time instant or during some time interval"* or also *"find the k-closest objects with respect to a given point at a given time instant"*. Trajectory-based queries involve topology of trajectories (e.g., overlap and disjoint) and related information (e.g., speed, area, and heading) that can be derived from the combination of time and

space. An example of such queries would be *"find objects or trajectories that satisfy a spatial predicate (eg., leave or enter a region) at a particular time instant or time interval"*. There also exist combined queries addressing information of particular objects: *"Where was object X at a particular time instant or time interval?"*. In all previous queries, the results are the individual trajectories that satisfy the query constraints.

In many applications, aggregated trajectories must be analyzed in the collection, as an individual trajectory does not represent any representative information. In this context, we can further distinguish range- from trajectory-based queries. Range queries impose constraints in terms of a spatial location and temporal interval. Examples of these queries are *"to retrieve the number of distinct trajectories that intersect a spatial region or spatial location (stop) at a given time instant or time interval"*, *"retrieve the number of distinct trajectories that start at a particular location (stop) or in a region and/or end in another particular location of region"*, *"retrieve the number of trajectories that follow a path"*, and *"retrieve the top-k locations (stops) or regions with the larger number of trajectories that intersect at a given time instant or time interval"*. Trajectory-based queries require not only to use the spatio-temporal points of trajectories but also the sequence of these points. Examples of such queries are to *"find the number of trajectories that are heading (not necessarily ending at) to a spatial location during a time interval"*, *"find the destination of trajectories that are passing through a region during a time interval"* or also *"find the number of starting locations of trajectories that go or pass through a region during a time interval"*.

2.3 Trajectory indexing

Literature on spatial trajectory indexing can be categorized by the nature of the trajectories: they can be either constrained to a network or in free space (often called moving objects). While there are well-known queries for indexes that work on moving objects in free space [PJT00], the network-constrained trajectory indexes cover more diverse querying needs, as different networks involve different kinds of challenges (as in vehicular road network vs public transportation network), and also because the intended application may vary (analyzing trip-chaining patterns vs number of passengers within a time window). A comprehensive review on indexing methods can be found in [PT14, Chapter 4], to which we shall expand in the rest of this section to mention the most remarkable indexes and some new developments.

2.3.1 Free trajectory indexing

Several adaptations of the *R-Tree* [Gut84] are widely used for the indexing of moving objects. The most common approach is to integrate the temporal dimension in the R-Tree itself, as found in the *STR-Tree* and the *TB-Tree* from [PJT00]. Another common approach is to complement the R-Tree with another similar structure, as has been done for the *MV3R-tree* [TP01], where an Historical *R-Tree* [NS98] was used to partition on the temporal dimension.

Generally, even for very large collections of trajectories, the spatial dimensions are more bounded than the temporal dimension, which can grow indefinitely. For this reason, even for free trajectories, temporal dimension is generally more selective than spatial dimensions.

This observation was heavily exploited in the subsequent works, such as *SETI* from [CEP03], where the space is partitioned statically, while trajectories are indexed by their temporal dimension using a one dimensional R-Tree, allowing it to grow dynamically.

Recently a framework based on Apache Spark was developed called *UlTraMan* [DCG+18], that supports different kinds of partitioning schemes for large collections of trajectories to answer range, kNN, or aggregation queries, allowing to repartition the dataset to maximize the query efficiency for a given query type. Although UlTraMan has been tested over network-constrained trajectories, it does not appear to be exploiting network information in any way, hence its inclusion in this category.

In the field of compact data structures, an index called *GraCT* [BGBNP19] has been developed. It is based on representing snapshots at regular time intervals using k^2*-Trees* [BLN09] and keeping movement logs for individual trajectories. Such movement logs are grammar compressed by using *RePair* [LM00]. Because of this, GraCT is a self-indexed compact representation that supports spatio-temporal range and nearest-neighbor queries, as well as allowing for direct access to the trajectory information.

2.3.2 Network-constrained trajectory indexing

There are also numerous approaches that use R-Tree-based indexes for trajectories that are constrained to an underlying network, aiming to decrease the redundancy in the representation by separating the representation into levels. Examples of such indexes include the *FNR-Tree* [Fre03], where the network elements are indexed with a 2D R-Tree. Every network element at a leaf node of this R-Tree points to a 1D R-Tree that is used to index the start and end of the time intervals at which moving objects pass through that edge of the network. As such, the FNR-Tree is only capable of recording simple movements where the edges are assumed to be traversed at constant speed. These limitations are addressed in [dAG05], where the authors propose the *MON-Tree*, where the moving objects were indexed in two dimensional R-Trees (the dimensions being edge position vs time). More recent alternatives, such as the *TMN-Tree* [CSU10], integrate B^+*-Trees*, which have proven to be more space and time efficient for indexing the temporal dimension. Alternatively, in [RRS18] a compact representation of time intervals is proposed using two bitvectors, that can be combined with those R-Tree-based indexes to increase the efficiency of the temporal filtering. Refer to [JSR17] for a quick comparison of these R-Tree-based indexes.

As a competitive alternative to these R-Tree-based indexes, *PARINET* [SPZO+11] builds spatial partitions from the trajectories based on their distribution and network topology, and uses a B^+-Tree to index the trajectories in each partition by time intervals. Although candidate trajectories must be filtered in memory during queries, PARINET is highly efficient in practice as its partitioning strategy minimizes the number of disk accesses needed.

Another relevant representation is described in [KPTT14]. Their proposed index, *NETTRA*, was designed to efficiently solve a specific kind of queries called *Strict Path Queries (SPQ)*, built on a traditional RDBMS with B^+-Tree indexes. Another distinctive feature of NETTRA is its treatment of network-constrained trajectories as textual information, which allows to apply string matching techniques such as fingerprinting functions to determine what trajectories have similar paths on their traversed edges. This close equivalence between trajectories and strings has been further exploited by

Geodabs [CG18], where both the spatial distribution and sequence information are taken into account for finding trajectories by similarity with fingerprinting. These text-based approaches are sometimes tangled with works on the topic of semantic trajectories such as [ADWK+17].

A recent compact data structure named *CiNCT* has been proposed in [KTXI18], where trajectories are encoded into a string, that is used to build an FM-index [FM00] with a Huffman-shaped Wavelet Tree [FGNV09]. To further save space, the string is constructed with relative movement labels instead of absolute edges, with an auxiliary structure that represents a network graph built from the input trajectories themselves. While this structure excels at pattern-matching and path extraction in vehicular networks (such as the streets of a city), it cannot be applied to our problems in public transportation networks, where network demand and aggregated travel patterns have to be analyzed. Note that finding out which trajectories traversed on a specific path of sequential street segments does not provide us much relevant information for our needs.

Part I

Compact representation for trajectories over transportation networks

Chapter 3

Previous concepts

This chapter presents a brief discussion of the compact data structures used in our proposed representations, and it gives a background of the underlying concepts that we will be working with for the rest of this part. For a more in-depth guide on compact data structures, the reader may refer to [Nav16].

A reader who is already familiar with compact data structures is still encouraged to read Section 3.1, where the Summed Area Table (SAT) is introduced, which is initially unrelated to compact data structures, and also Section 3.4, where we discuss some less known variants and operations of the Wavelet Tree (WT), that we make use of in our contributions.

3.1 Summed Area Table

The *Summed Area Table (SAT)* was first introduced in computer graphics [Cro84] to speed up the mipmapping process, where given a texture image represented as a series of bidimensional matrices of numbers (usually three matrices of integers, one for each color channel) we are interested in finding the average color of any arbitrary rectangle within the image. This operation is most often used to reduce the rendering time for distant polygons where a pixel on the target screen may correspond to several texture pixels (texels), and also for anisotropic filtering, in order to improve the visual quality of polygons that are projected in an oblique angle.

With the most direct representation of a matrix as an array of values, we are required to compute the average value of a rectangle $[(a, b), (a + h, b + w)]$ as:

$$\overline{M}[a..a+h, b..b+w] \leftarrow \frac{\sum_{i=0}^{h}\sum_{j=0}^{w} M[i+a, j+b]}{(h+1)(w+1)}$$

Note that the summation has a time complexity of $O(hw)$, which would make this operation quite expensive for real-time rendering applications. In order to decrease the complexity of these calculations, the SAT precomputes the summations of M in a matrix

I, of the same size, where $I[a,b] \leftarrow \sum_{i=0}^{a}\sum_{j=0}^{b} M[i,j]$. That is, each cell of I is the sum of the values within the rectangle spanning from the origin of the matrix to the position of the cell. An example of a SAT is shown in Figure 3.1.

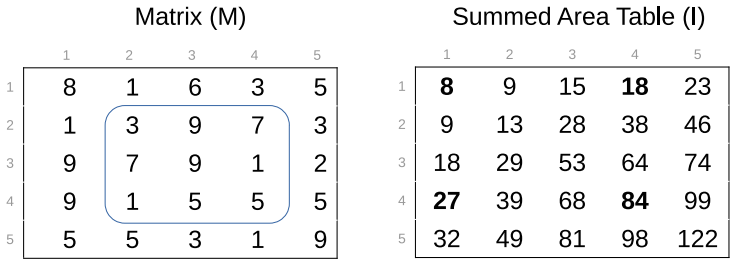


Figure 3.1: An example of a Summed Area Table (right) built over a matrix (left).

Having I, we can compute the average value of a rectangle in $O(1)$ operations as:

$$\overline{M}[a..a+h, b..b+w] \leftarrow \frac{I[a+h,b+w] - I[a+h,b-1] - I[a-1,b+w] + I[a-1,b-1]}{(h+1)(w+1)}$$

In our example from Figure 3.1, to calculate the sum of the delimited $3x3$ region in the matrix M, we simply operate over the four terms in **bold** from I, obtaining $\sum_{i=2}^{4}\sum_{j=2}^{4} M[i,j] = I[4,4] - I[4,1] - I[1,4] + I[1,1] = 84 - 27 - 18 + 8 = 47$.

While with this representation we do not need to keep the original matrix,[1] the improved computational efficiency comes at the expense of having to represent larger numbers than the original values.

3.2 Entropy coding

Given an information source (such as a text) that provides symbols from an alphabet $[1..\sigma]$ with a probability of $0 \leq p_i \leq 1$ for each symbol, where $\sum_i p_i = 1$, the goal of an entropy coder is to exploit these probabilities in order to achieve compression by assigning shorter codes to the most frequent symbols, and longer codes to the less frequent ones. In the work that is considered as the foundation of information theory [Sha48], Claude Shannon defined a concept called *entropy*, which is closely related to the probabilities we are discussing, and used it to prove that when encoding the symbols in binary, the optimal length in bits for

[1] As accessing a single cell can be viewed as a special case of the computation of an average where $h = 0, w = 0$.

each code is of $l_i = \frac{1}{\log_2 p_i} = -\log_2 p_i$ bits, and the entropy of an information source S is calculated as $H_0(S) = -\sum_i^\sigma p_i l_i = -\sum_i^\sigma p_i \log_2 p_i$.

For any compression technique relying on using variable-length codes, it is necessary for the codes to be unambiguous: there can be no two codes C_i, C_j that, when concatenated, could be interpreted as another code . Additionally, it is computationally very useful for those codes to be also prefix-free, meaning that there can be no code that is the prefix part of another code ($C_i \neq C_j\{0|1\}^a,\ \forall i,j \in [1..\sigma], i \neq j, a \in [0..\infty)$). This will allow us to unambiguously interpret the symbol right after the bits of C_i, without having to determine if it could be the prefix part of some other longer code C_j.

The Huffman coding, introduced in [Huf52], is a coding algorithm that produces optimal[2] prefix-free codes based on the frequencies of each symbol. The ideas of the Huffman coding have been widely implemented in numerous compression algorithms and codecs since its inception, where the most notable examples are DEFLATE (PKZIP, GZIP), JPEG and MPEG.

One less known variation of the Huffman codes are the Hu-Tucker codes [HT71], which aim to provide codes that preserve the same lexical order as the original symbols, meaning that for any two symbols s_i, s_j, it holds that $s_i \prec s_j \iff C_i \prec C_j$. This comes at the expense of at most one extra bit per code over Huffman on average.[3]

3.3 Bitvectors

A vast amount of works in compact data structures involves the use of bitvectors, both compressed and uncompressed. A bitvector $B[1..n]$ is a sequence of n bits, for which the following operations are expected to be supported:

- $\text{rank}_1(B, i)$ is the number of set bits in $B[1..i]$. Alternatively, $\text{rank}_0(B, i) \leftarrow i - \text{rank}_1(B, i)$. Consequently, it also holds that the bit from the position i can be retrieved as $B[i] = \text{rank}_1(B, i) - \text{rank}_1(B, i-1)$, with a special case of $\text{rank}_1(B, 0) = 0$.
- $\text{select}_1(B, i)$ is the position in $[1..n]$ where the i-th 1 occurs. Therefore, $\text{rank}_1(B, \text{select}_1(B, i)) = i$. An equivalent version for 0 bits may be defined as $\text{select}_0(B, i)$, although, unlike rank_0, there does not exist a direct way of obtaining it from the previously defined operations.

Example 3.1: Given a bitvector $B = 011001$, it holds that $\text{rank}_1(B, 1) = 0$, $\text{rank}_1(B, 2) = 1$, $\text{rank}_1(B, 3) = 2, \text{rank}_1(B, 4) = 2$. Furthermore, $B[3] = \text{rank}_1(B, 3) - \text{rank}_1(B, 2) = 1$ and $\text{rank}_0(B, 3) = 3 - \text{rank}_1(B, 3) = 1$.

We can also say that $\text{select}_1(B, 2) = 3$ and $\text{select}_1(B, 3) = 6$, thus holding that $\text{rank}_1(B, \text{select}_1(B, 2)) = \text{rank}_1(B, 3) = 2$ and $\text{rank}_1(B, \text{select}_1(B, 3)) = \text{rank}_1(B, 6) = 3$. Additionally, $\text{select}_0(B, 2) = 4$ as $\text{rank}_0(B, \text{select}_0(B, 2)) = \text{rank}_0(B, 4) = 2$. □

All these operations can be supported in $O(1)$ time with $o(n)$ extra bits [Jac89, Mun96]. Additionally, there exist several techniques for compressing these bitvectors based on

[2] That is, with lengths as close to $-\log_2 p_i$ as possible with a whole number of bits per symbol.

[3] Being L_h and L_{ht} the average codeword length of Huffman coding and Hu-Tucker codes for S respectively, it holds: $H_0(S) \leq L_h \leq H_0(S) + 1$ and $H_0(S) \leq L_{ht} \leq H_0 + 2(S)$ (see [CT06] (pages 122-123), or [Hor77, GM59]).

their statistical properties or the arrangement of the bits. In this work we will use a representation that excels in efficiently solving select_1 operations for sparse bitvectors (with a low number of set bits in proportion to the bitvector size) introduced in [OS07], and also another representation that works particularly well with non uniformly distributed bitvectors, thus exploiting their entropy to require only $nH_0(B) + o(n)$ bits[4] [RRR02].

3.4 Wavelet Tree and Wavelet Matrix

When working with a sequence $S[1..n]$ built over an alphabet $[1..\sigma]$, we can extend the definitions of rank and select from Section 3.3 to work over any symbol $a \in [1..\sigma]$ instead of bits, resulting in the following operations:

- $\text{rank}_a(S, i)$ gives the number of occurrences of the symbol a in $S[1..i]$.
- $\text{select}_a(S, i)$ gives the position in $[1..n]$ where the i-th a occurs.

The Wavelet Tree (WT) [GGV03] represents S with a balanced binary tree, with σ leaves, one for each symbol. Every internal node v represents the range of symbols $[\alpha_v..\omega_v] \subseteq [1..\sigma]$ in the same order as they appear in S, where the root node corresponds to all the n symbols from the alphabet $[1..\sigma]$ appearing in S. Its left child v_l will only represent the n_l symbols that fall in the range $[1..\frac{\sigma}{2})$, preserving the order that they had in v_0, while the right child will represent the other n_r symbols from $[\frac{\sigma}{2}..\sigma]$, holding that $n_l + n_r = n$. This is built recursively until a leaf v_a is reached, with will correspond to only one symbol $a \in [1..\sigma]$, with $n_a = \text{rank}_a(S, n)$ times. An example WT can be found in Figure 3.2.

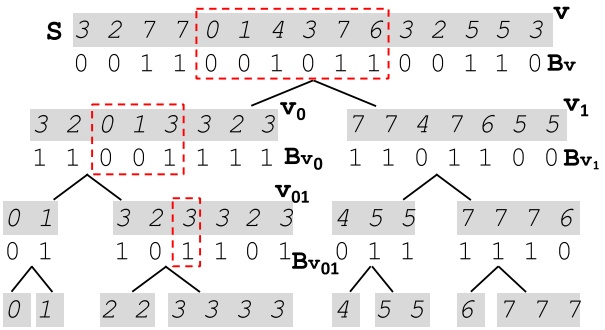


Figure 3.2: An example Wavelet Tree of four levels.

For each node v, these symbols from the alphabet $[\alpha_v..\omega_v]$ are represented implicitly using a bitvector $B_v[1..n_v]$, where $B[i] = 0$ when the symbol a represented in the position

[4] Being n_1 the number of set bits in B and $n_0 = n - n_1$, then $H_0(B) = \dfrac{n_1 \log \frac{n}{n_1} + n_0 \log \frac{n}{n_0}}{n} \leq 1$.

i should belong to the left child ($a < \frac{\alpha_v + \omega_v}{2}$), while $B[i] = 1$ means it should belong to the right child ($a \geq \frac{\alpha_v + \omega_v}{2}$). By making use of rank and select operations over these bitvectors, we are able to recover the value of any $S[i]$, and also to answer rank_a and select_a for any $a \in [1..\sigma]$ in $O(\log \sigma)$ without the need of storing the original sequence. As there are n bits per level and $\log \sigma$ internal levels (the nodes from the leaf level do not have bitvectors), we could represent all the bitvectors in $n \log \sigma$ bits, which is equivalent to the space it would take to represent S as an array of fixed length integers.[5] However, we also need some auxiliary structures for rank_1 and select_1, which will require $o(n \log \sigma)$ bits and also store a pointer for every one of the $2\sigma - 1$ nodes. Therefore, the total size of this representation amounts to $n \log \sigma + o(n \log \sigma) + O(\sigma \log n)$.

In order to access the value of $S[i]$, we must start by looking into $B_v[i]$ from the root node. If $B_v[i] = 0$, we traverse to the position $B_{v_0}[\text{rank}_0(B_v, i)]$ of the bitvector of the left child. Conversely, when $B_v[i] = 1$, we traverse to the bitvector of the right child at $B_{v_1}[\text{rank}_1(B_v, i)]$. In both cases, we recurse until a leaf is reached, which will unambiguously correspond to the symbol a, thus we determine that $S[i] = a$.

Example 3.2: In the WT from Figure 3.2, if we wanted to retrieve the value of $S[2]$, we would start by looking into the bit $B_v[2] = 0$, meaning that the we have to traverse to the node v_0 and look into the bit $B_{v_0}[\text{rank}_0(B_v, 2)] = B_{v_0}[2] = 1$, leading us to the node v_{01} at $B_{v_{01}}[\text{rank}_1(B_{v_0}, 2)] = B_{v_{01}}[2] = 0$. The left node is a leaf node belonging to the symbol 2, so $S[2] = 2$ (namely, the first occurrence of 2 in S, as $\text{rank}_0(B_{v_{01}}, 2) = 1$). □

We can also solve rank_a by traversing the tree from top to bottom with the rank operation on the bitvectors: knowing that the binary representation (and thus also the path in the WT) of a symbol will allow us to use rank_1 or rank_0 in each level i according to the i-th bit of our code. In the Example 3.4, knowing that the binary representation of 2 is 010, we can calculate $\text{rank}_2(S, 2)$ (or any other position) following the same order of operations: a rank_0, a rank_1 and one final rank_0 to determine the position of the leaf node, which will give away rank_2. It is also possible to calculate $\text{select}_a(S, i)$ with a bottom-up traversal of the tree, by starting at the i-th position from the leaf belonging to a, and applying select_0 and select_1 on the bitvectors of the parent nodes, following the reversed binary code of a. Eventually, a position $S[j] = a$ will be reached such that it will contain the i-th occurrence of a in S, consequently obtaining that $j = \text{select}(S, i)$.

A more complex operation that we have found very useful in our work is the operation $\text{count}_{a,b}(S, i, j)$, first described in [GNP12], which counts the number of occurrences of the symbols in $[a..b]$ within $S[i..j]$. While it is equivalent to $\sum_{k=a}^{b} \text{rank}_k(S, j) - \text{rank}_k(S, i)$, it can be solved more efficiently by doing two simultaneous top-down traversals, as described for rank_a. Starting at the root node v we calculate $\text{rank}_0(B_v, i)$ and $\text{rank}_0(B_v, j)$ to find out the limits in the left node for the codes within $[a..b]$ that start with 0, and also $\text{rank}_1(B_v, i)$ and $\text{rank}_1(B_v, j)$ for those codes starting by 1. If all the codes in $[a..b]$ start with either a zero or a one, we will only compute the corresponding rank. After that, we recurse for each node, where on the left we set $i \leftarrow \text{rank}_0(B_v, i)$, $j \leftarrow \text{rank}_0(B_v, j)$ and we only consider

[5] When σ is not a power of two, $\log \sigma$ would need to be rounded up for fixed length integers, while for the WT there will be some leaves at level $h - 1$, being h the height of the tree, and its total size will depend on the frequency of those symbols ($\lfloor n \log \sigma \rfloor < \sum_i n_i < \lceil n \log \sigma \rceil$).

the range of symbols $[a..b']$ that can be represented by that node ($b' \leq \omega_{v_0}$)). The same is true for the recursion on the right, where $i \leftarrow \text{rank}_1(B_v, i)$, $j \leftarrow \text{rank}_1(B_v, j)$ and $\alpha_{v_1} \leq a'$. Therefore, we compute recursively:

$$\begin{aligned}\text{count}_{a,b}(v, i, j) = &\text{count}_{a,b'}(v_0, \text{rank}_0(B_v, i), \text{rank}_0(B_v, j)) \\ + &\ \text{count}_{a',b}(v_1, \text{rank}_1(B_v, i), \text{rank}_1(B_v, j))\end{aligned}$$

The recursion on each node v stops when $a \leq \alpha_v$ and $\omega_v \leq b$, in which case the count for that node is reported as $j - i + 1$. Finally, all the counts are summed and the total amount of occurrences is obtained. While it may look as if the worst case would imply traversing all σ internal nodes, it is avoided by stopping the recursion when $a \leq \alpha_v$ and $\omega_v \leq b$, thus requiring to visit only $O(\log \sigma)$ nodes.[6] In the WT from Figure 3.2, we have marked in red the ranges considered for solving $\text{count}_{3,3}(S, 5, 10)$. Additionally, when the subsequence $S[i..j]$ is sorted, we can define $[l..r] \leftarrow \text{count}^{LR}_{a,b}(S, i, j)$, which returns the upper and lower limits $S[l..r]$ of the occurrences of the symbols $[a..b]$. Naturally, $S[l..r] \subseteq S[i..j]$.

3.4.1 Hu-Tucker Wavelet Tree

A straightforward way of reducing the size of a WT is to use compressed bitvectors, as discussed in [CN08], allowing to represent a WT in $nH_0(S) + o(n \log \sigma) + O(\sigma \log n)$ bits [GGV03]. There is, however, a different approach to achieve similar space requirements is to use a prefix-free variable-length encoding for the symbols. For example, Huffman code [Huf52] can be used to build a Huffman-Shaped WT [FGNV09], where the tree is not balanced anymore, as the level of each leaf v_a will be the number of bits for the Huffman code of a, which will depend on the frequency of a in S. The size reduces to $n(H_0(S)+1) + o(n(H_0(S)+1)) + O(\sigma \log n)$,[7] while average time becomes $O(H_0(S))$ for rank_a and select_a (the worst-case time is still $O(\log \sigma)$ [BN13]). By using compressed bitvectors [CN08] space can be even further reduced to $nH_0(S) + o(n(H_0(S) + 1)) + O(\sigma \log n)$. Unfortunately, the Huffman codes (including *canonical Huffman*) assigned to lexicographically adjacent symbols do not maintain that lexicographic order, and it is not possible to have a $O(\log \sigma)$ bound for $\text{count}_{a,b}(S, i, j)$ anymore.

In order to support $\text{count}_{a,b}(S, i, j)$ more efficiently, Hu-Tucker codes [HT71] can be used instead. While the compression achieved by a Hu-Tucker Wavelet Tree (HTWT) [BN09] degreades slightly with respect to using Huffman coding, yielding a bound of $n(H_0(S) + 2) + o(n(H_0(S) + 1)) + O(\sigma \log n)$,[8] the codes for adjacent symbols are lexicographically contiguous. Therefore, we can guarantee a bound of $O(\log \sigma)$ for $\text{count}_{a,b}(S, i, j)$ again. An example of a HTWT in practice can be found in Figure 4.4.

3.4.2 Wavelet Matrix

For large alphabets, the size of the WT is affected by the term $O(\sigma \log n)$. A pointerless WT [CN08] permits to remove[9] that term by concatenating all the bitvectors level-wise

[6] In fact, the best case is $cnt_{1,\sigma}(S, i, j) = j - i + 1$, which is solved without traversing at all.

[7] $O(\sigma \log n)$ term includes both the tree pointers and the size of the Huffman model.

[8] This can be reduced to $nH_0(S) + o(n(H_0(S) + 1)) + O(\sigma \log n)$ by using compressed bitvectors as well.

[9] In a pointerless Huffman-shaped WT a term $O(\sigma \log \log n)$ still remains due to the need for storing the canonical Huffman model.

and computing the values of the pointers during the WT traversals. The operations on a pointerless WT have the same time complexity but become slower in practice.

By reorganizing the nodes in each level of a pointerless WT, the Wavelet Matrix (WM) [CNO15] obtains the same space requirements, yet its performance is very close to that of the regular WT with pointers. Figure 3.3 contains an example of a Wavelet Matrix (WM), representing the same sequence as in Figure 3.2.

Figure 3.3: An example of a Wavelet Matrix with three levels. A conceptual fourth level was omitted since it does not contain a bitvector.

As in the WT, the i-th level stores the i-th bits of the encoded symbols. A single bitvector B_i is kept for each level. In the first level, B_1 stores the 1-st bit of the encoding of the symbols in the order of the original sequence S. From there on, at level i, symbols are reordered according to the $(i-1)$-th bit of their encoding; that is, according to the bit they had in the previous level. The symbols whose encoding had a zero at position $i-1$ must be arranged before those that had a one. After that, the relative order from the previous level is maintained. That is, if a symbol a occurred before some other symbol b, and the $(i-1)$-th bit of their encoding coincides, then a will precede b at level i.

If we simply keep track of the number of zeros at each level $z_l \leftarrow \text{rank}_0(B_l, n)$, we can easily see that the symbol with the k-th zero at level $i-1$ is mapped at position k within B_i, whereas the symbol with the j-th one at level $i-1$ is mapped at position $z_l + j$ within B_i. This avoids the need for pointers, enabling to retain the same time complexity of the WT operations, including $\text{count}_{a,b}(S, i, j)$. For implementation details see [CNO15, Ord16].

Example 3.3: To find out the symbol at $S[8]$, we start by observing that $B_1[8] = 0$ and $\text{rank}_0(B_1, 8) = 5$. We move to the next level where we check position 5; we see that $B_2[5] = 1$ and $\text{rank}_1(B_2, 5) = 3$. We move to next level and check position $3 + z_2 = 3 + 5 = 8$, where we finally see $B_3[8] = 1$. Therefore, we have decoded the bits **011** that correspond to the symbol $S[8] = 3$. □

To reduce the space needs of the WM we could use compressed bitvectors as for the WT. Yet, compressing the WM by giving it either a Huffman or Hu-Tucker shape is not possible as the reordering of the WM would lead to the existence of gaps in the bitvectors that would ruin the process of tracking symbols during traversals. To overcome this issue, an optimal

Huffman-based coding was specifically developed for wavelet matrices [CNO15, FGM+16]. This allows to obtain space similar to that of a pointerless Huffman-shaped WT but with faster rank_a and select_a operations. Unfortunately, since the encodings of consecutive symbols do not keep the same order, $\text{count}_{a,b}(S, i, j)$ is no longer supported in $O(\log \sigma)$ time.

3.5 Compressed Suffix Array

Given a sequence $S[1..n]$[10] built over an alphabet Σ of length σ, the *suffix array* $A[1..n]$ is built over S [MM93] as a permutation of the positions $i \in [1..n]$ of all the suffixes $S[i..n]$, so that $S[A[i]..n] \prec S[A[i+1]..n]$ for all $1 \leq i < n$ Because A contains all the suffixes of S in lexicographic order, we can use this structure to search for any pattern $P[1..m]$ in time $O(m \log n)$ by simply performing binary searches for the range $A[l..r]$ that contains pointers to all the positions in S where P occurs. We can find an example of an suffix array A in Figure 3.4. Any pattern $P[1..m]$ (such as *ana*) can be delimited by a range $A[l..r]$ (for "*ana*" it is $A[3..4]$), where the limits l and r can be found with binary searches due to the suffixes being sorted.

	1	2	3	4	5	6	7
S =	b	a	n	a	n	a	$
A =	7	6	4	2	1	5	3
Ψ =	5	1	6	7	4	2	3
D =	1	1	0	0	1	1	0

	1	2	3	4
V =	$	a	b	n

Figure 3.4: All the structures involved in constructing a Compressed Suffix Array over the sequence $S = banana\$$. Note that S and A do not need to be stored.

A straightforward enhancement to avoid storing the original string S is to set up a vocabulary array $V[1..\sigma']$, with all the different symbols from Σ appearing in S,[11] and a bitvector $D[1..n]$ aligned to A so that $D[1] = 1$ and $D[i] = 1 \iff S[A[i-1]] \neq S[A[i]]$ for all the other $i \in [2..n]$. This means that D marks with a 1 the beginning of a range of suffixes pointed from A such that the first symbol of those suffixes coincides. With D, keeping S is no longer needed since $S[A[i]] = V[\text{rank}_1(D, i)]$.

We can also replace A as described in Sadakane's Compressed Suffix Array (CSA) [Sad03], using another permutation $\Psi[1..n]$ defined in [GV00], where $\Psi[i] = A^{-1}[A[i]+1]$.

[10] For convention, we establish that $S[n]$ must contain a terminator symbol \$ that must be lexicographically smaller than any of the other symbols in $S[1..n-1]$.

[11] Note that $\sigma' \leq \sigma$) since some of the symbols from Σ may never occur in S.

That is, for every $A[i] = k$ and $A[j] = k + 1$, $\Psi[i] = j$. The special case when $A[i] = n$[12] is handled as $\Psi[i] = A^{-1}[1]$, making a cycle. Therefore, the Compressed Suffix Array (CSA) is formed by Ψ, D, and V, which are sufficient to simulate binary searches of the interval $A[l..r]$ for the occurrences of P without the need of accessing A nor S. In this work, we define that operation as $[l..r] \leftarrow \text{bsearch}(\Psi, P)$. The symbol $S[A[i]]$ pointed by $A[i]$ can be obtained as $V[\text{rank}_1(D, i)]$. We can also easily obtain the following symbol from the source sequence $S[A[i] + 1]$ as $V[\text{rank}_1(D, \Psi[i])]$, $S[A[i] + 2]$ can be obtained as $V[\text{rank}_1(D, \Psi[\Psi[i]])]$, and so on.

Although an uncompressed Ψ would have the same space requirements as A, it is highly compressible, since it is formed by σ strictly increasing subsequences. By using δ-codes of the gaps (differences of each value with respect to the previous one) it is possible to compress Ψ to around the zero-order entropy of S [Sad03], with $nH_0(S) + O(n \log \log \sigma)$ bits. In [NM07] it has been further proved that Ψ can be split into at most $nH_k + \sigma^k$ (for any k) *runs* of consecutive values so that the differences within those runs are always 1. This allows for a combination of δ-coding of gaps with run-length encoding (of 1-*runs*) to achieve a higher-order compression of Ψ without further intervention. In addition, to maintain fast random access to Ψ, absolute samples at regular intervals (every t_Ψ entries) are kept. Parameter t_Ψ implies a space/time trade-off. Larger values lead to better compression of Ψ but slow down access time to non-sampled $\Psi[i]$ values.

In [FBN+12], the authors have adapted Sadakane's CSA to deal with large (integer-based) alphabets and created the *integer-based CSA* (*iCSA*). They also showed that, in their scenario (natural language text indexing), the best compression of Ψ was obtained by combining differential encoding of runs with Huffman and run-length encoding.

[12] $i = 1$ if we use the unique terminator \$.

Chapter 4

Representations for trajectories over urban streets

As explained in Section 1.2, we have identified two contexts for public transportation systems according to their networks, which can be based on urban streets or public transportation. While the proposed structure in this chapter is capable of operating within both contexts, it does not take public transportation elements (routes and vehicles) into account. This leads to a more redundant representation than the ones later proposed in Chapter 5, which are more adequate for public transportation networks.

The work in this chapter proposes a new structure named Compact Trip Representation (CTR) that answers counting-based queries and uses compact self-indexed data structures to represent the large amount of trajectories in compact space. CTR combines two well-known data structures. The first one, initially designed for the representation of strings, is the CSA. The second one is the WT. With these two structures, CTR is able to efficiently resolve queries over trajectory patterns in any dimension (spatial, temporal or, combining both structures, spatio-temporal).

In Section 4.1 we present our scenario, and describe the simplified data model used to represent user trajectories. The most interesting queries that are defined for this scenario are also discussed. Then, Section 4.2 describes the internals of CTR and Section 4.3 is devoted to explain how query operations are solved in CTR. After that, experiments are included in Section 4.4.

We experimentally tested our proposal using two sets of data representing trajectories over two different real public transportation systems. Our results are promising because the representation uses around 50% of its original size and answers most of our spatial, temporal, and spatio-temporal queries within $1-1000$ microseconds. No experimental comparisons with classical spatial or spatio-temporal index structures were possible, because none of them were designed to answer the types of queries in this work. Our approach can be considered as a proof of concept that opens new application domains for the use of

well-known compact data structures such as the CSA and the WT, creating a new strategy for exploiting trajectories represented in a self-indexed way.

4.1 Description

Given a transportation network, whether it is based on urban streets or public transportation, we work with a representation of the network that is based on a directed graph. For an urban street network, a **node** represents a road segment delimited by intersections, where two nodes are connected by an edge if it is possible (i.e. legally allowed). This allows to accurately describe a trajectory by sequentially listing the road segments that were traversed, while minimizing the redundancy.

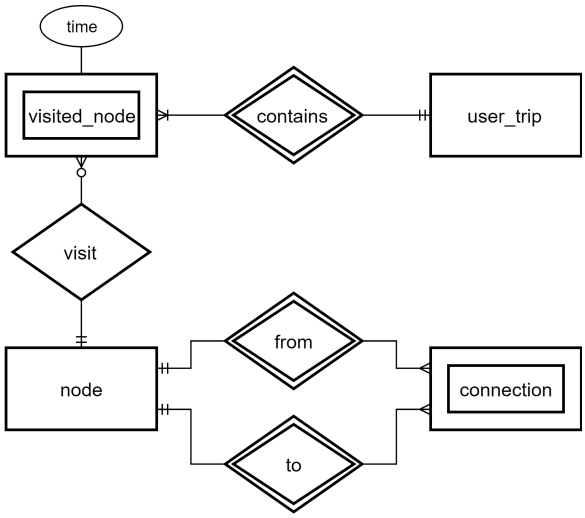


Figure 4.1: An ER diagram representing the model for user trajectories for CTR.

Figure 4.1 contains the entity-relationship diagram of our network model, where nodes and connections define a directed graph, over which user trajectories can be conformed by sequentially visiting the nodes. The order in which these nodes are visited is implicitly defined by the time, meaning that it is necessary to somehow represent that time for the visited nodes of each trajectory.

To make the use of CSA possible, we define a trip or trajectory of a moving object over a network as the temporally-ordered sequence of the nodes the trajectory traverses. An integer $s_i \in S$ is assigned to each node such that a trajectory is a sequence (string) of consecutively nodes visited by a single user. Note that this representation avoids the cost of storing coordinates to represent the location users pass through during a trajectory. It is just enough to identify the stops or nodes and when necessary to map these nodes to

geographic locations. Moreover, when the underlying network is formed by street segments, we do not specify at which part of the segment did the trajectory start or finished: we consider such level of detail irrelevant for traffic analysis, as it can be effectively made on a street-segment level.

We then build a CSA, over the concatenation of these strings (trajectories), with some adaptations for this specific application. In addition, we discretize the time in periods of fixed duration (i.e. timeline split into 5-minute intervals) and each time segment is identified by an integer $t_i \in I$. In this way, it is possible to store the times when trajectories reach each node by associating the corresponding t_i with each node in each trajectory. The sequence of times for all the nodes within a trajectory is self-indexed with a WT to efficiently answer temporal and spatio-temporal queries.

Among other types of queries, in this work we focus on the following counting queries, which to the best of our knowledge have not been addressed by previous proposals. In general terms, we define two general queries, number-of-trips queries and top-k queries, upon which we apply spatial, temporal, or spatio-temporal constraint when useful.

(a) *Number-of-trips queries.* This is a general type of queries that counts the number of distinct trajectories. When applying spatial, temporal, or spatio-temporal constraints, it can specialized in the following queries:

1. Pure spatial queries:
   - *Number of trips starting at node X (`start_X`).*
   - *Number of trips ending at node X (`end_X`).*
   - *Number of trips starting at X and ending at Y (`from_X_to_Y`).*
   - *Number of trips using or passing through node X. Can also be seen as the average load of the node X. (`load_X`)*
2. Spatio-temporal queries:
   - *Number of trips starting at node X during time interval $[t_1..t_2]$ ($\texttt{start\_X}_T$).*
   - *Number of trips ending at node X during the time interval $[t_1..t_2]$ ($\texttt{end\_X}_T$).*
   - *Number of trips starting at X and ending at Y occurring during time interval $[t_1..t_2]$ ($\texttt{from\_X\_to\_Y}_T$).* This type of queries is further classified into:
     (i) $\texttt{from\_X\_to\_Y}_T$ with strong semantics ($\texttt{from\_X\_to\_Y}_{Ts}$), which considers trajectories that completely occur within interval $[t_1..t_2]$.
     (ii) $\texttt{from\_X\_to\_Y}_T$ with weak semantics ($\texttt{from\_X\_to\_Y}_{Tw}$), which considers trajectories whose life time overlap $[t_1..t_2]$.
   - *Number of trips using node X during the time interval $[t_1..t_2]$. Can also be seen as the average load of the node X within a given time interval. ($\texttt{load\_X}_T$).*
3. Pure temporal queries:
   - *Number of trips starting during the time interval $[t_1..t_2]$ (`start_T`).*
   - *Total usage (load) of network nodes during the time interval $[t_1..t_2]$ (`load_T`).*
   - *Number of trips performed within the time interval $[t_1..t_2]$ (`trip_T`).*

(b) *Top-k queries.* In this type of queries we want to retrieve the k nodes with the highest number of trips. In this case, depending on having a temporal constraint or not we include the following queries:

1. Pure spatial *Top-k* queries:
   - *Top-k most used nodes (`top_K`)*, that returns the nodes with the largest number of trips passing through.
   - *Top-k most used nodes to start a trip (`top_K`$_s$)*, that returns the nodes with the largest number of trips that start at that node.
2. Spatio-temporal *Top-k* queries:
   - *Top-k most used nodes during time interval* $[t_1..t_2]$*(`top_K`$_T$)*, that returns the nodes with the largest number of trips passing through within time interval $[t_1..t_2]$.
   - *Top-k most used nodes to start a trip during time interval* $[t_1..t_2]$*(`top_K`$_{Ts}$)*, that returns the nodes with the largest number of trips starting there within time interval $[t_1..t_2]$ at that node.

4.2 Structures

To support the queries seen in Section 4.1, we need to represent the spatial and temporal components of our collection of user trips that is coherent with the network model described. Therefore, we will proceed to detail how each trip is described, before we show how that description is implemented in our compact data structures.

If we consider a network $\mathcal{N}$ with a set of nodes S, we can see a dataset of trips $\mathcal{T}$ over $\mathcal{N}$ as a set of trips, where for each trip $\mathcal{T}_i \in \mathcal{T}$, we represent a list with the n_i temporary-ordered nodes it traverses and the corresponding timestamps: $\mathcal{T} = \{\langle(s_1^i, s_2^i, \ldots, s_{n_i}^i), (t_1^i, t_2^i, \ldots, t_{n_i}^i)\rangle\}$, $i \in [1..|\mathcal{T}|]$, $s_j^i \in S$, and $t_x^i \leq t_y^i, \forall x < y$. Note that every node in the network can be identified with an integer ID $s_j^i \in S$ and that, if we are interested in analyzing the usage patterns of the network, we will also be interested in discretizing time into time intervals (i.e. 5-min, 30-min intervals). Therefore, we will have $|I|$ different time intervals that can also be identified with an integer ID ($t_j^i \in I$).

The size of the time interval is a parameter for the time-discretizing process that can be adjusted to fit the required precision in each domain. For example, in a public transportation network where we could have data including five years of trips, one possibility would be to divide that five-years period into 10-minute intervals hence obtaining a vocabulary of $|I| = 5 \times 365 \times 24 \times 60/10 = 262,800$ different intervals. Other possibility would be to use cyclically annual 10-minute periods resulting in $|I| = 262,800/5 = 52,560$. However, in public transportation networks, queries such as *"Number of trips using the stop X on May 10 between 9:15 and 10:00"* may be not as useful as queries such as *"Number of trips using stop X on Sundays between 9:15 and 10:00"*. For this reason, CTR can adapt how the time component is encoded depending on the queries that the system must answer.

Example 4.1: Figure 4.2 shows a network that contains $|S| = 10$ nodes numbered from 1 to 10. Over that network we have six trips ($|\mathcal{T}| = 6$), and, for each of them, we indicate the sequence of nodes it traverses and the time when the trip goes through those nodes. If we discretize time into 5-minute intervals, starting at 08:00h, and ending at

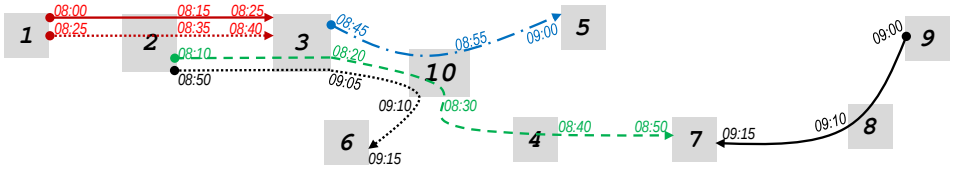


Figure 4.2: A set of trips over a network with 10 nodes.

9:20h, we will have have $|I| = 16$ different time intervals. Any timestamp within interval $[08:00, 08:05)$ will be assigned time-code 0, those within $[08:05, 08:10)$ code 1, and so on until times within $[09:15, 09:20)$ that are given time-code 15. Therefore, our dataset of trips will be: $\mathcal{T}$: $\{\langle(\mathbf{1},\mathbf{2},\mathbf{3}), (5, 7, 8)\rangle, \langle(\mathbf{2},\mathbf{3},\mathbf{10},\mathbf{6}), (10, 13, 14, 15)\rangle, \langle(\mathbf{1},\mathbf{2},\mathbf{3}), (0, 3, 5)\rangle, \langle(\mathbf{2},\mathbf{3},\mathbf{10},\mathbf{4},\mathbf{7}), (2, 4, 6, 8, 10)\rangle, \langle(\mathbf{3},\mathbf{10},\mathbf{5}), (9, 11, 12)\rangle, \langle(\mathbf{9},\mathbf{8},\mathbf{7}), (12, 14, 15)\rangle\}$, where bold numbers indicate node IDs and slanted ones indicate times. □

In CTR we represent both the spatial and the temporal component of the trips using well-known self-indexing structures in order to provide both a compact representation and the ability to perform fast indexed searches at query time. In Section 4.2.1 we focus on the spatial component and discuss how we adapted CSA to deal with trips. We also show how we support spatial queries. Then, in Section 4.2.2 we show that the times, which are kept aligned with the spatial component of the trips, can be handled with a WT-based representation. Actually we study two alternatives (a HTWT and a WM) and show how temporal and spatio-temporal (Section 4.3.3) queries are supported by CTR.

4.2.1 Spatial component using a CSA

We use a slightly adapted CSA to represent the spatial component of our dataset of trips within CTR. However, we must perform some preprocessing on each trip $\mathcal{T}_i \in \mathcal{T}$ before building a CSA on it. Initially, we sort the trips by their first node (s_1^i), then by the last node (s_n^i), then by the starting time (t_1^i), and finally, by its second node (s_2^i), third node (s_3^i), and successive nodes (i.e. the trips are sorted by the key $s_1, s_n, t_1, s_{2..n-1}$. Note that the start time (t_1^i) of the trip does not belong to the spatial component, but it is nevertheless used for the sorting.[1]

Following with Example 4.2, after sorting the trips in $\mathcal{T}$ with the criteria above, our sorted dataset $\mathcal{T}^s$ would look like: $\mathcal{T}^s$: $\{\langle(\mathbf{1},\mathbf{2},\mathbf{3}), (0, 3, 5)\rangle, \langle(\mathbf{1},\mathbf{2},\mathbf{3}), (5, 7, 8)\rangle, \langle(\mathbf{2},\mathbf{3},\mathbf{10},\mathbf{6}), (10, 13, 14, 15)\rangle, \langle(\mathbf{2},\mathbf{3},\mathbf{10},\mathbf{4},\mathbf{7}), (2, 4, 6, 8, 10)\rangle, \langle(\mathbf{3},\mathbf{10},\mathbf{5}), (9, 11, 12)\rangle, \langle(\mathbf{9},\mathbf{8},\mathbf{7}), (12, 14, 15)\rangle\}$. Note that $(\mathbf{2},\mathbf{3},\mathbf{10},\mathbf{6})$ appears before $(\mathbf{2},\mathbf{3},\mathbf{10},\mathbf{4},\mathbf{7})$ because during the sorting process we compare $(\mathbf{2},\mathbf{6}, 2, \mathbf{3},\mathbf{10},\mathbf{6})$ with $(\mathbf{2},\mathbf{7}, 10, \mathbf{3},\mathbf{10},\mathbf{4},\mathbf{7})$; that is, we compare the starting nodes (**2** and **2**) and then the ending nodes (**6** and **7**). If needed (not in this example) we would have also compared the slanted values (*2* and *10*) that are the starting times of the trips, and finally the rest of nodes ($\mathbf{3},\mathbf{10},\mathbf{6}$ and $\mathbf{3},\mathbf{10},\mathbf{4},\mathbf{7}$).

[1] This initial sorting of the trips will allow us to answer some useful queries very efficiently (i.e., count trips starting at node X and ending at node Y).

Similarly, the two trips containing nodes $(\mathbf{1}, \mathbf{2}, \mathbf{3})$ are sorted by the starting times (0 and 5).

In a second step, we enlarge all the trips $\mathcal{T}_i^s \in \mathcal{T}^s$ with a fictitious terminator-node $\$_i$ whose timestamp is set to that of the initial node of the trip. We choose terminators such that $\$_i \prec \$_j, \forall i < j$; that is, the lexicographic value of $\$_i$ is smaller for smaller i values. In addition, the lexicographic value of any terminator must be lower than the ID of any node in a trip. Therefore, an enlarged trip $\mathcal{T}_i^s$ would become $\mathcal{T}_i' = \langle (s_1^i, s_2^i, \ldots, s_{n_i}^i, \$_\mathbf{i}), (t_1^i, t_2^i, \ldots, t_{n_i}^i, \mathbf{t_1^i}) \rangle$.

The next step involves concatenating the codes s_j^i and $\$_i$ of the spatial components of our trips and to add an extra trailing terminator $\$_0$ to create a sequence $Text[1..n]$.[2] $\$_0$ must be lexicographically smaller than any other entry (then it also holds $\$_0 \prec \$_i$, $\forall i \in [1..|\mathcal{T}|]$). In the top part of Figure 4.3, we can see array $Text$ for the running example, as well as the corresponding time-IDs that are regarded in sequence $Icode$ ($Time$ shows the original times).

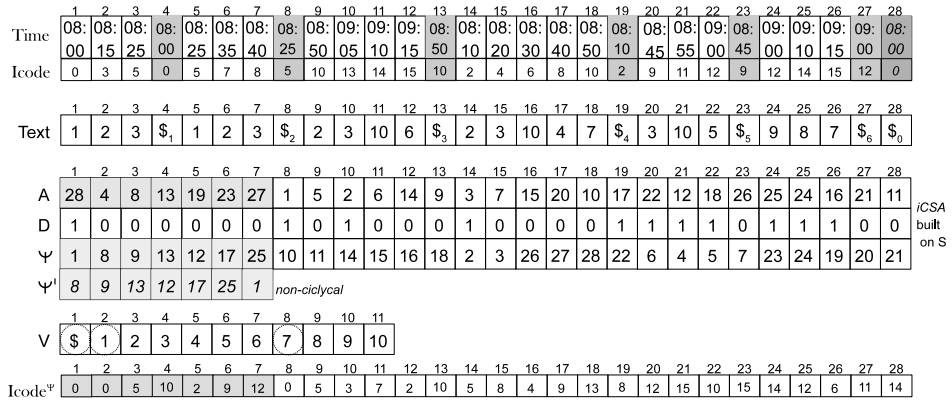


Figure 4.3: Structures involved in the creation of a CTR.

Finally, we build a CSA on top of $Text$ to obtain a self-indexed representation of the spatial component in CTR. Figure 4.3 depicts the structures Ψ and D used by the CSA built over $Text$. There is also a vocabulary V containing a \$ symbol and the different node IDs in lexicographic order.

Note that the use of different values $\$_i$ as terminators ensures that our sorting criteria are kept even if we follow the standard suffix-sort procedure[3] required to build suffix array A during the creation of CSA. Yet, when we finish that process, we can replace all these $1 + |\mathcal{T}|$ terminators $\$_i$ by a unique \$. This is the reason why there is only one \$ symbol in V.

[2] By definition, it must hold that $n = |\mathcal{T}| + 1 + \sum_{i=1}^{|\mathcal{T}|} n_i$.

[3] The suffix $Text[i..n]$ is compared with the suffix $Text[j..n]$.

Although they are not needed in CTR, we show also suffix array A and Ψ' for clarity reasons in Figure 4.3. Ψ' contains the first entries of Ψ from a regular CSA, whereas we introduced a small variation in CTR for entries $\Psi[1..(|\mathcal{T}|+1)]$. Displaying both Ψ and Ψ' helps us to better illustrate our process to build Ψ. For example, $A[8] = 1$ points to the first node of the first trip $S[1]$. $\Psi[8] = 10$ and $A[10] = 2$ point to the second node. $\Psi[10] = 14$ and $A[14] = 3$ point to the third node. $\Psi[14] = 2$ and $A[2] = 4$ point to the ending $\$_1$ of the first trip. Therefore, in the standard CSA, $\Psi'[2] = 9$ and $A[9] = 5$ point to the first node of the second trip. However, in CTR, $\Psi[2] = 8$ and $A[8] = 1$ point to the first node of the first trip. With this small change, subsequent applications of Ψ will allow us to cyclically traverse the nodes of the trip instead of accessing the following entries of $Text$.

Another interesting property arises from the use of a cyclical Ψ on trips, and from using trip terminators. Since the first entries in $\Psi[2..|\mathcal{T}|+1]$ correspond the \$ symbols that mark the end of each trip in $Text$ (remember that $\Psi[1]$ corresponds the $\$_0$), we can see that the j-th node of the i-th trip can be obtained as $V[\text{rank}_1(D, \Psi^j[i+1])]$, (where $\Psi^3[x] = \Psi[\Psi[\Psi[x]]]$). This property makes it very simple to find starting nodes for any trip. For example, if we focus on the shaded area $\Psi[2..7]$, we can find the ending terminator $\$_4$ of the fourth trip at the 5-th position (because the first $\$_0$ corresponds to the final \$ at $S[28]$). Therefore, its starting node can be found as $V[\text{rank}_1(D, \Psi[4+1])]$. Since $\Psi[5] = 12$ and $\text{rank}_1(D, 12) = 3$, the starting node is $V[3] = \mathbf{2}$. For illustration purposes note that it would correspond to $Text[A[12]]$. By applying Ψ again, the next node of that trip would be obtained by computing $\Psi[12] = 16$, $\text{rank}_1(D, 16) = 4$, and accessing $V[4] = \mathbf{3}$ (that is, we have obtained $V[\text{rank}_1(D, \Psi[\Psi[4+1]])] = \mathbf{3}$, and so on.

Regarding the space requirements of the CSA in CTR, we can expect to obtain a good compressibility due to the structure of the network, and the fact that trips that start in a given node or simply those going through that node will probably share the same sequence of "next" nodes. This will lead us to obtaining many *runs* in Ψ ([NM07]), and consequently good compression could be expected.

4.2.1.1 Implementation details

In our implementation of CSA, we used the $iCSA$[4] from [FBN+12] briefly discussed in Section 3.5. Yet, we introduced some small modifications:

- The construction of the Suffix Array A is done with *SA-IS* algorithm [NZC11].[5] In comparison with the *qsufsort* algorithm[6] [LS07] used in the original $iCSA$, it achieves a linear time construction and a lower extra working space. Refer to [MPPP19] for an up to date comparison among the modern Suffix Array construction methods.
- In $iCSA$, a plain representation for bitvector D was used, with additional structures to support rank_1 in constant time using $(0.375 \times n$ bits). With that structure, they could solve select in $O(\log n)$ time (yet they did not actually needed solving select in $iCSA$). In our CSA, we have used the *SDArray* from [OS07] to represent D. It provides a very good compression for sparse bitvectors, as well as constant-time select_1 operation.

[4] `http://vios.dc.fi.udc.es/indexing`

[5] We have used the improved implementation by Yuta Mori, available at `https://sites.google.com/site/yuta256/sais`

[6] `http://www.larsson.dogma.net/research.html`

- In [FBN+12], bsearch(Ψ, P) operation was implemented with a simple binary search over Ψ rather than using the backward-search optimization proposed in the original CSA [Sad03]. In our proposals, we used backward search since it led to a much lower performance degradation at query time when a sparse sampling of Ψ was used.

4.2.2 Time intervals representations

In this section, we focus on the temporal component associated with each node of the enlarged trips $\mathcal{T}'_i$ in our dataset, previously described in Section 4.2.1. Recall that in Figure 4.3, sequence $Time$ contains the discretized time intervals associated with each visited node in a trip, and $Icode$ a possible encoding of times. In CTR we focus on the values in $Icode$, yet, since $Text$ is not kept anymore in CTR, we reorganize the values in $Icode$ to keep them aligned with Ψ rather than with $Text$. Those values are represented within array $Icode^{\Psi}$ in Figure 4.3. For example, we can see that $Icode^{\Psi}[4]$ corresponds with $Icode[A[4]] = 10$, $Icode^{\Psi}[15]$ corresponds with $Icode[A[15]] = 8$, and so on. Conveniently, the first $|\mathcal{T}| + 1$ entries of $Icode^{\Psi}$ will contain the time interval codes for the start of each trip, as each $\$_i$ was originally aligned with a copy of the first time interval t_1^i of each trip.

Aiming at having a compact representation of $Icode^{\Psi}$ while permitting fast access and resolution of range-based queries (that we could use to search for trips within a given time interval), we have considered two WT-based alternatives from the ones presented in Section 3.4:

- A Wavelet Tree using variable-length Hu-Tucker codes (HTWT). Recall this is the WT variant that permits to compress the original symbols with variable-length codes and still supports $\text{count}_{a,b}(S, i, j)$ operation in $O(log\ \sigma)$ time. Since Hu-Tucker coding assigns shorter codes to the most frequent symbols, the compression of our HTWT is highly dependent of the distribution of frequencies for the $Icode^{\Psi}$. Yet, if our trips represent movements of single users in a transportation network, we could expect to observe two or more periods corresponding to rush hours within a single day (see Section 4.4.1). This would lead to obtaining a skewed distribution of the frequencies for the symbols in $Icode^{\Psi}$, and consequently, we could expect to have better compression than if we used a balanced WT. The expected number of bits of our HTWT is $nH_0(Icode^{\Psi})$.
- A balanced Wavelet Matrix (WM). As we have shown in Section 3.4, the WM is typically the most compact uncompressed variant of WT and it is faster than a pointerless WT. This is the reason why we chose a balanced WM instead of a balanced WT as this second alternative. Recall that, $Icode^{\Psi}$ contains n symbols, and each symbol can be encoded with $\log |I|$ bits, hence the balanced WM can be seen as a matrix of $n \log |I|$ bits.

In Figure 4.4, we show both the WM and the HTWT built on top of $Icode^{\Psi}$ from Figure 4.3. The binary code-assignment to the source symbols $t_i \in I$ and that obtained after applying Hu-Tucker encoding algorithm [HT71] are also included in the figure.

Since the most useful operation of these two structures for our application is, by far, $\text{count}_{a,b}(Icode^{\Psi}, i, j)$, we will proceed to explain in detail how the efficiency of that operation has influenced our choice of structures. Just as proven in Section 3.4 with WT, both HTWT and WM implement the $\text{count}_{a,b}(Icode^{\Psi}, i, j)$ operation in $O(\log\ \sigma)$ time

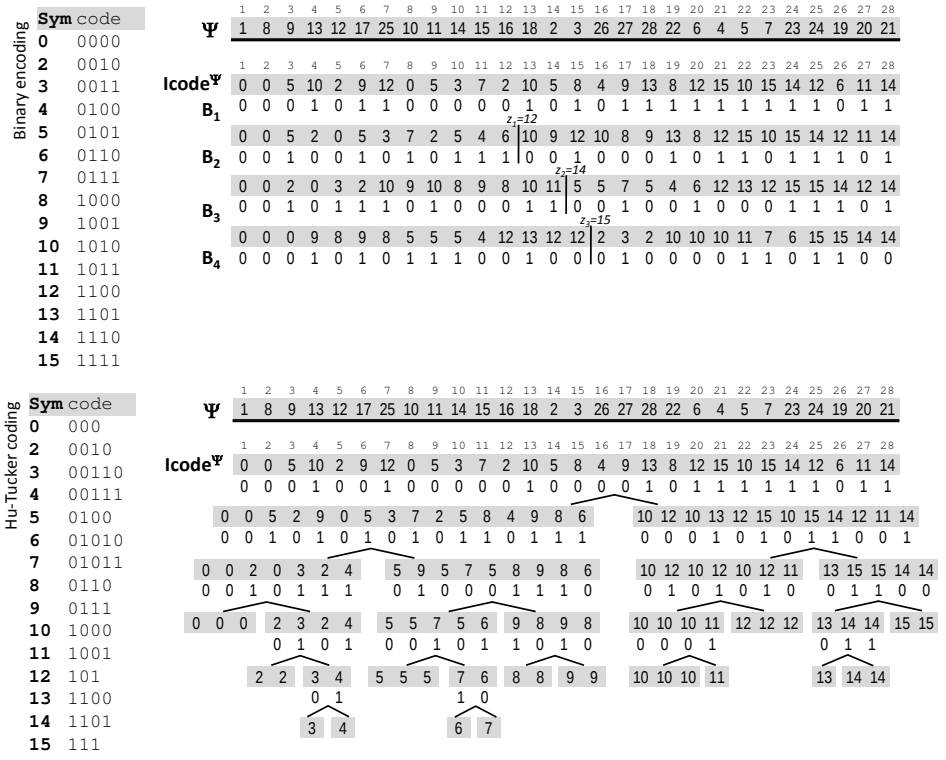


Figure 4.4: Balanced WM (top) and HTWT (bottom).

on average. This is easy to prove for HTWT as each node will contain entries from a lexicographically contiguous subrange from the alphabet Σ, making the same properties seen for WT hold. The main difference is that Hu-Tucker codes will reshape the tree, making the leaves containing the least frequent symbols deeper than the leaves with the more frequent symbols, which may theoretically produce a tree of height $\sigma - 1$,[7] which will obviously affect the worst-case performance, in exchange of an improved compression ratio.

The complexity may seem initially harder to analyze for WM, as its "nodes" are delimited implicitly: the symbols with a code starting with a 0 will find their second bit in an implicit node delimited by the subrange $B_2[1..z]$ (for some $1 \leq z < n$), while the symbols with a code starting with a 1 will correspond to $B_2[z+1..n]$. Generalizing this idea, we can assert that symbols with the same *context* of α starting bits in their codes will find their next $\alpha + 1$ bit in some subrange $B_{\alpha+1}[i..j]$ with $1 \leq i \leq j \leq n$. Therefore, the same properties from WT can be exploited for a $\text{count}_{a,b}(Icode^{\Psi}, i, j)$ operation in WM as

[7] Assume a vocabulary $\Sigma = \{s_1, ..., s_\sigma\}$, where the probability of appearance of the symbol s_i in the text S turned out to be 2^{-i}.

well, allowing us to solve it in $O(\log \sigma)$ worst-case time.

While it would also be possible to build WM using canonical Huffman codes, it is unfortunately impossible to guarantee the $O(\log \sigma)$ bound on $\text{count}_{a,b}(Icode^{\Psi}, i, j)$ on such WM, as the symbols lose their original lexicographic proximity, meaning that symbols which would appear on the same node of WT due to the common most significant bits in their original codes, would get different Huffman codes based on their frequency of appearance in the text, ending up in separate nodes. On the other hand, to the best of our knowledge, there is no practical way of building WM with Hu-Tucker codes.

4.2.2.1 Implementation details

As we discussed in Section 3.4, both HTWT and WM are built over bitvectors that require support for rank and select operations. In our implementations we included two alternative bitvector representations avaliable at *libcds* library:[8]

- A plain bitvector based on [Mun96] named *RG* with additional structures to support rank in constant time (select in logaritmic time). *RG* includes a sampling parameter ($factor$) that we set to value 32. In this case, our bitvector *RG* uses $n(1 + 1/32)$ bits. That is, we tune *RG* to use a sparse sampling.
- A compressed RRR bitvector [RRR02]. The *RRR* implementation includes a sampling parameter that we tune to values 32, 64, and 128. Higher sampling values typically achieve better compression.

In advance, when presenting results for HTWT and WM we will consider the four bitvector configurations above. Regarding our implementations of WM and HTWT, note that we reused the same implementation of WM from [CNO15], and we created our custom HTWT implementation, paying special focus at solving $\text{count}_{a,b}(Icode^{\Psi}, i, j)$ efficiently.

4.3 Algorithms

In this section, we discuss how our previously described structures can solve the queries proposed in Section 4.1. We also include a brief complexity analysis for some of the cases.

4.3.1 Spatial queries

With the CSA structure described for representing the spatial component of the trips, the following queries can be solved.

- *Number of trips starting at node X (`start_X`).* Because Ψ was cyclically built in such a way that every \$ symbol is followed by the first node of its trip, this query is solved by $[l..r] \leftarrow \text{bsearch}(\Psi, \$X)$ over the CSA, which results on a binary search for the pattern $\$X$ over the section $\Psi[2..|\mathcal{T}|]$ corresponding to \$ symbols. Then $r - l + 1$ gives the number of trips starting at X. Applying the backward search algorithm for CSA, this query involves two select_1 operations over D in order to delimit the

[8] `https://github.com/fclaude/libcds`

region $\Psi[l_x..r_x]$ for X[9] and one binary search in the \$ region to find the subrange for $\$X$. Since select_1 on D are $O(1)$, the temporal complexity of this query is $O(\log |\mathcal{T}|)$, omitting a constant factor due to the compression of Ψ.

- *Number of trips ending at node X (`end_X`).* In a similar way to the previous query, this one can be answered with $\text{bsearch}(\Psi, X\$)$. It will require four select_1 operations (still $O(1)$) and a binary search over the X region, giving a total worst-case complexity of $O(\log(n - |\mathcal{T}|)) \subset O(\log n)$.
- *Number of trips starting at X and ending at Y (`from_X_to_Y`).* Combining both ideas from above, and thanks to the cyclical construction of Ψ, this query is solved using $\text{bsearch}(\Psi, Y\$X)$. As it requires two binary searches, the first one to delimit the $\$X$ region and the second one to find the entries within Y that point to that $\$X$ region, the overall complexity is $O(\log n)$.
- *Number of trips using node X (`load_X`).* Even though we could solve this query with $\text{bsearch}(\Psi, X)$, it is more efficient to solve it by directly operating on D, by finding the region $\Psi[l..r]$ for X and calculating $r - l + 1$. All the operations involved are $O(1)$.
- *Top-k most used nodes (`top_K`).* We provide two possible solutions for this query named: a sequential and a binary-partition approach:
  - The *sequential* ($\texttt{top\_K}_{seq}$) approach is the simpler alternative to compute the k most used nodes. The idea is to apply $\text{select}_1(D, i)$ operations sequentially for every $i \in [2..|V|$ to compute the frequency of each node and to return the k nodes with highest frequency. We use a min-heap that is initialized with the first k nodes, and for every node s from $k + 1$ to $|V|$, we compare its frequency with that of the minimum node (the root) from the heap. In case the frequency of s is higher, the root of the heap is replaced by s and then moved down to comply with the heap ordering. At the end of the process, the heap will contain the top-k most used nodes $\langle p_1\ ,\ p_2,\ \ldots,\ p_k\rangle$, which can be sorted with the heapsort algorithm if needed. Note that, since $|S| = |V| - 1$, this approach will perform $|S|$ select_1 operations on D, as well as up to $|S|$ insertions in the heap of size k, thus having an overall complexity of $O(|S| \log k)$.
  - The *binary-partition* ($\texttt{top\_K}_{bin}$) approach takes advantage of the skewed frequency distribution for the nodes that trips traverse. Working over D and V, we recursively split D into two segments after each iteration. If possible, we leave the same number of different nodes in each side of the partition. Initially, we start considering the range in $D[l..r] \leftarrow D[\text{select}_1(D, 2)..n] = D[|\mathcal{T}| + 2..n]$ which corresponds to the nodes that appear in V from positions $i = 2$ to $j = |V|$.[10] We use a priority queue that is initialized as $Q \leftarrow (\langle i, j\rangle, \langle l, r\rangle)$. Then, assuming $m = i + \frac{j-i+1}{2}$ and $q = \text{select}_1(D, m)$, we create two partitions $D[l..q-1]$ and $D[q..r]$, which correspond respectively to the nodes in $V[i..m-1]$

[9] Assuming that X is at position p in the vocabulary V of CTR ($V[p] = X$), its region in $\Psi[l_x..r_x]$ is obtained as $l_x \leftarrow \text{select}_1(D, p)$, $r_x \leftarrow \text{select}_1(D, p + 1)$. If p is the last entry in V, we set $r_x \leftarrow n$.

[10] We skip the \$ at the first entry of V and its corresponding entries in D; that is, $D[1..\,\text{select}_1(D, 2) - 1]$.

and $V[m..j]$. These segments created after the partitioning step are pushed into Q. The pseudocode can be found in Algorithm 1.

The priority of each segment in Q is directly the size of its range in D $(r-l+1)$. When a segment extracted from Q represents the instance of only one node $((\langle i,j\rangle,\langle l,r\rangle)$, with $i=j)$, that node is returned as a result of the top-k algorithm (we return $V[i]$). The algorithm stops when the first k nodes are found.

For example, when searching for the top-1 most used nodes in the example from Figure 4.3, Q is initialized with the segment [8..28], corresponding to nodes from 1 to 10 (positions from 2 to 11 in V). Note that the entries of D from 1 to 7 and $V[1]$ represent the \$ symbol. Since it is not an actual node, it must be skipped. Then [8..28] is split producing the segments [8..20] for nodes 1 to 5 ($V[2..6]$) and [21..28] for nodes 6 to 10 ($V[7..11]$). After three more iterations, we extract $(\langle 3,3\rangle,\langle 14,18\rangle)$, hence obtaining the segment [14..18] for the single node 3 (position 4 in V), concluding that the *Top-1 most used node* is $\mathbf{3}=V[4]$ with a frequency equal to $5=18-14$.

Even though the worst-case complexity of this approach is $O(|S|\log|S|)$, which can be expected when the distribution of nodes is uniform, it can perform considerably better than the sequential approach with a skewed distribution and a small k, as will be experimentally proven in Section 4.4.3.1.

- *Top-k most used nodes to start a trip (`top_K`$_s$).* Both `top_K` approaches above can be adapted for answering `top_K`$_s$. However, unlike its simpler variant, it requires performing bsearch$(\Psi,\$X)$ over Ψ (rather than a select$_1$ on D) at each iteration, hence increasing the temporal complexity of the operation.

  The implementation of the linear approach is straightforward. The binary-partition approach differs slightly from Algorithm 1: in *line 3* we have insert $(\langle 2,|V|\rangle,\langle 2,z+1\rangle)$ into Q, and we replace *line 12* with $[x..y]\leftarrow$ bsearch$(\Psi,\$V[m])$; $q\leftarrow x$. This increases the temporal complexity of `top_K`$_s$ by a factor of $O(\log n)$ over the complexities discussed for the original `top_K` queries.

4.3.2 Temporal queries

With either one of the described alternatives (HTWT or WM) to represent time intervals we can answer the following purely temporal queries:

- *Number of trips starting during the time interval $[t_1..t_2]$ (`start_T`).* Since we keep the starting time of each trip within $Icode^{\Psi}[2..|\mathcal{T}|+1]$, we can efficiently solve this query by simply computing $\text{count}_{t_1,t_2}(Icode^{\Psi},2,|\mathcal{T}|+1)$ in $O(\log|I|)$ time.
- *Total usage of network nodes during the time interval $[t_1..t_2]$ (`load_T`).* This query can be seem as the sum of the number of trips that traversed each network node during $[t_1..t_2]$. We can solve this query by computing $\text{count}_{t_1,t_2}(Icode^{\Psi},|\mathcal{T}|+2,n)$, in $O(\log|I|)$ time.
- *Number of trips performed during the time interval $[t_1..t_2]$ (`trip_T`).* This is also an interesting query that measures the actual number of trips started or completed within the queried time interval. To solve this query we could compute `trip_T` by subtracting

```
1  Function GetTopK_most_used_nodes(k):
       Data: number k
       Result: topK nodes
2      Q ← new PriorityQueue();
3      Push(Q,(⟨2, |V|⟩, ⟨select_1(D, 2), n⟩));
4      current_k ← 0;
5      while current_k < k do
6          (⟨i, j⟩, ⟨l, r⟩) ← Pop(Q);
7          if i =j then
8              topK[current_k] ← V[i];
9              current_k ← current_k + 1;
10         else
11             m ← i + (j−i+1)/2;
12             q ← select_1(D, m + 1);
13             Push(Q,(⟨i, m − 1⟩, ⟨l, q − 1⟩));
14             Push(Q,(⟨m, j⟩, ⟨q, r⟩));
15         end
16     end
17     return topK;
```

Algorithm 1: Algorithm *Top-k most used nodes* using binary-partition approach.

the number of trips that started after t_2 and the number of trips that ended before t_1 from the total number of trips ($|\mathcal{T}| - \texttt{start\_T}(t_2+1, |I|) - \texttt{end\_T}(1, t_1-1)$). However, recall that $Icode^{\Psi}[2..|\mathcal{T}|+1]$ has the starting time of each trip, but we do not keep their ending time. We could solve $\texttt{end\_T}(1, t_1-1)$ by taking the first node (X) of each trip starting before t_1, then applying Ψ until reaching the ending node (Y), and finally getting the ending time of that trip associated to node Y. However, this would be rather inefficient. A possible solution to efficiently solve $\texttt{end\_T}(1, t_1-1)$, would require to augment our temporal component, in parallel with $Icode^{\Psi}[2..|\mathcal{T}|+1]$, with another WT-based representation of the ending times for our $|\mathcal{T}|$ trips. This would permit to report the number of trips ending before t_1 as $\text{count}^{LR}_{0,t_1-1}(Icode^{\Psi}, 2, |\mathcal{T}|+1)$, but would increase the overall size of CTR. Yet, note that even without keeping ending-times, we could provide rather accurate estimations of `trip_T` for a system administrator. For example, using `load_T` to compute the number of times each trip went through any node during the time interval $[t_1..t_2]$, and dividing that value by the average nodes per trip. Another good estimation can also be obtained with $\texttt{start\_T}(t_1, t_2)$.

4.3.3 Spatio-temporal queries

Apart from the pure spatial and temporal queries discussed in the previous sections, we can combine both the self-indexed spatial and temporal components from CTR to answer spatio-temporal queries. The idea is to restrict spatial queries to a time interval $[t_1..t_2]$. An example of this type of query is to return the *number of trips starting at node X that occurred between t_1 and t_2*, which we can solve by first finding the range in the CSA of the trips starting in X and then relying on the *count* operation in the HTWT (or WM). The following spatio-temporal queries can be solved by CTR:

- *Number of trips starting at node X during time interval $[t_1..t_2]$ (`start_`X_T).* Recall that in the time sequence we also included timestamps associated with the area of \$-symbols in $\Psi[1..|\mathcal{T}|+1]$. Particularly, for each \$ at $\Psi[i+1]$, we keep the time of the first node of its trip $\mathcal{T}_i$. Therefore, we can perform $[l..r] \leftarrow \text{bsearch}(\Psi, \$X)$ as in a regular spatial query to find the range $\Psi[l..r]$ ($[l..r] \subseteq [2..|\mathcal{T}|+1]$) that corresponds to \$ symbols that end a trip which started at node X. Then, since the time sequence $Icode^{\Psi}$ (represented with either a HTWT or WM) is aligned with Ψ, we can filter out those trips that started within $[t_1..t_2]$ performing operation $\text{count}_{t_1,t_2}(Icode^{\Psi}, l, r)$. Because both bsearch and count are used, the time complexity for the whole query is $O(\log n + \log |I|)$. In Figure 4.5 (steps ① and ②) we illustrate the steps involved.

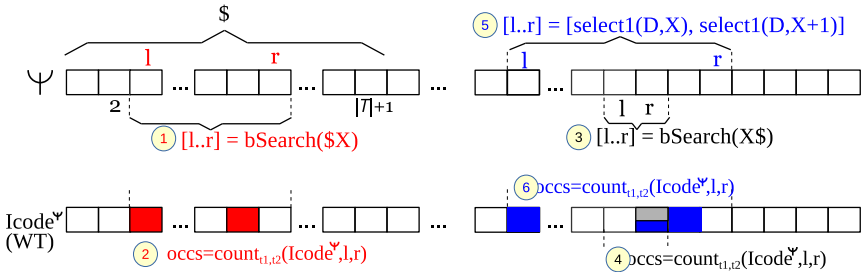


Figure 4.5: Trips staring at, ending at, or using node X during time interval $[t_1..t_2]$.

- *Number of trips ending at node X during the time interval $[t_1..t_2]$ (`end_`X_T).* As above, we initially perform the spatial query $[l..r] \leftarrow \text{bsearch}(X\$)$ to obtain the range in $\Psi[l..r]$ that corresponds to the pattern $X\$$ (trips ending at node X). Then, we use $\text{count}_{t_1,t_2}(Icode^{\Psi}, l, r)$ operation to count how many of those trips match the temporal constraint. See steps ③ and ④ in Figure 4.5.
- *Number of trips using node X during the time interval $[t_1..t_2]$ (`load_`X_T).* As in the corresponding spatial query, the range $\Psi[l..r]$ is obtained with two select_1 operations on D. Finally, $\text{count}_{t_1,t_2}(Icode^{\Psi}, l, r)$ finds the occurrences within the time interval $[t_1..t_2]$, thus solving this query in $O(log|I|)$ time. See steps ⑤ and ⑥ in Figure 4.5.
- *Number of trips starting at X and ending at Y occurring during time interval $[t_1..t_2]$ (`from_X_to_Y`$_T$).* We consider two different semantics. A query with *strong semantics* will obtain trips that start and end within $[t_1..t_2]$. Whereas, a query with *weak*

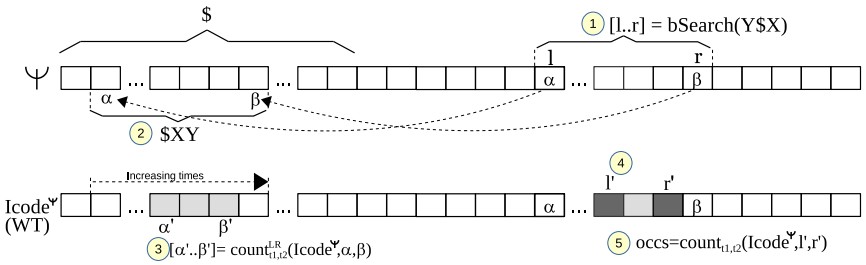


Figure 4.6: Trips staring at X and ending at Y during time interval $[t_1..t_2]$.

semantics will obtain trips whose time intervals overlap $[t_1..t_2]$ and, therefore, they could actually start before t_1 or end after t_2.

In Figure 4.6, we show the step-by-step process to solve this type of queries. As in a spatial query, we start by searching for the range $[l..r] \leftarrow \text{bsearch}(\Psi, Y\$X)$ corresponding to trips starting at Y and ending at X (*step-*①). Next, due to our sorting of trips, the range for $Y\$X$ in $\Psi[l..r]$ can be mapped to a continuous range $\Psi[\alpha..\beta]$ of the same size in the $\$X$ region of Ψ.[11] We compute $\alpha \leftarrow \Psi[l], \beta \leftarrow \alpha + r - l$ (*step-*②). Furthermore, note that the range for $\$XY$ preserves the same order as that for $Y\$X$.

At this point, since $Icode^{\Psi}$ was aligned with Ψ, we could check ending-time constraints within $Icode^{\Psi}[l..r]$ and starting-time constraints within $Icode^{\Psi}[\alpha..\beta]$ (recall we keep starting times associated with the corresponding $ of each trip). Note also that, due to our sorting (by starting-node, ending-node, starting-time,...) the times in $Icode^{\Psi}[\alpha..\beta]$ are increasing ($Icode^{\Psi}[i] \leq Icode^{\Psi}[i{+}1], \forall \alpha \leq i < \beta$), as long as they are within the region $\Psi[\alpha..\beta]$, which corresponds to trips with the same starting-node X and ending-node Y. Therefore, we can find the continuous subrange $[\alpha'..\beta'] \subseteq [\alpha..\beta]$ corresponding to trips that start within $[t_1..t_2]$ (*step-*③). This operation was defined as $\text{count}^{LR}_{a,b}(Icode^{\Psi}, i, j)$ in Section 3.4. Thus, that assuming $Icode^{\Psi}[\alpha..\beta]$ are increasing, $[\alpha'..\beta'] \leftarrow \text{count}^{LR}_{t_1,t_2}(Icode^{\Psi}, \alpha, \beta)$ would report the positions $[\alpha'..\beta'] \subseteq [\alpha..\beta]$ such that $\alpha' = argmin_x(Icode^{\Psi}[x] \geq t_1)$ and $\beta' = argmax_x(Icode^{\Psi}[x] \leq t_2)$.

The last step will differ on whether the query is implementing strong or weak semantics:

- *Strong semantics (`from_X_to_Y`$_{Ts}$).* Note that the subrange $[\alpha'..\beta']$ (containing trips starting within $[t_1..t_2]$) has a matching subrange $[l'..r'] = [l + \alpha' - \alpha..l + \beta' - \alpha] \subseteq [l..r]$ (*step-*④), where some of the ending times of these trips will fall inside $[t_1..t_2]$, allowing us to check the ending time constraint. By performing $\text{count}_{t_1,t_2}(Icode^{\Psi}, l', r']$ we get the final result (*step-*⑤). To sum up, answering

[11] As the trips were sorted by the key $s_1, s_n, t_1, s_{2..n-1}$, and we positioned each $\$_i$ according to the order of their trip $\mathcal{T}_i$, the region $Y\$X$ will be conveniently sorted by the key $s_n, \$, s_1, s_n, t_1, s_{2..n-1}$ within Ψ, thus delimiting an equivalent region $\Psi[\alpha..\beta]$ in $\$X$ where each entry corresponds to a trip that also ends in Y.

this query requires: one bsearch over Ψ (to find $[l..r]$), one access to Ψ to obtain α (since $\beta = \alpha + r - l$), one count^{LR} to find $[\alpha'..\beta']$, and one final count operation to count the valid ending times in $[l'..r']$, amounting in a total of $O(\log n + \log |I|)$ time.

- *Weak semantics (`from_X_to_Y`$_{Tw}$).* The size of $[\alpha'..\beta']$ $(\beta' - \alpha' + 1)$ is already a partial answer. To get the final result, we need to add also the occurrences of those trips starting before t_1 that end at t_1 or later, which can only exist if $\alpha < \alpha'$. To do so, we need to obtain $l' \leftarrow l + \alpha' - \alpha$ as done in `from_X_to_Y`$_{Ts}$, and compute $\text{count}_{t_1,|I|}(Icode^{\Psi}, l, l' - 1)$. This gives us the number of time instants in the range $[l..l')$ of $Icode^{\Psi}$ that fall inside $[t_1..|I|]$. That is, ending times equal or after t_1. Yet again, the time complexity for this query is $O(\log n + \log |I|)$.

- *Top-k most used nodes during time interval $[t_1..t_2]$ (`top_K`$_T$).* Both the sequential and binary-partition approaches discussed in Section 4.3.1 can be easily extended to support this query. The idea is that, when we add a node either to the min-heap or the priority-queue respectively, we compute its frequency within time interval $[t_1..t_2]$ (using count operation) rather than using its overall frequency.

  - In the *sequential approach (`top_K`$_{Tseq}$)*, given a node whose corresponding range in Ψ is $\Psi[l..r]$, we compute its frequency using $\text{count}_{t_1,t_2}(Icode^{\Psi}, l, r)$ instead of simply using $r - l + 1$. The rest of the process is exactly as discussed for the pure spatial `top_K`$_{seq}$ query. The time complexity is increased by a factor of $O(\log |I|)$ over the spatial `top_K`$_{seq}$ variant, resulting in $O(|S| \log k \log |I|)$.
  - In the *binary-partition approach (`top_K`$_{Tbin}$)*, we have to consider the priority of a given segment as the number of trips covered by that segment that occurred during $[t_1..t_2]$. Again, given a segment $[l..r]$ in Ψ we compute that priority as $p_{l..r} \leftarrow \text{count}_{t_1,t_2}(Icode^{\Psi}, l, r)$ instead of $p_{l..r} \leftarrow r - l + 1$. Apart from that, the only modifications that we must consider over the pure spatial `top_K`$_{bin}$ Algorithm 1 are: we replace *line 3* by $p_{l..r} \leftarrow \text{count}_{t_1,t_2}(Icode^{\Psi}, \text{select}_1(D, 2), n)$; $Q.push(\langle 2, |V| \rangle, \langle \text{select}_1(D, 2), n \rangle, \underline{p_{l..r}})$, and we replace *lines 12 and 13*, respectively, by $Q.push(\langle i, m - 1 \rangle, \langle l, q - 1 \rangle, \underline{\text{count}_{t_1,t_2}(Icode^{\Psi}, l, q - 1)})$ and $Q.push(\langle m, j \rangle, \langle q, r \rangle, \underline{\text{count}_{t_1,t_2}(Icode^{\Psi}, q, r)})$. Due to these modifications, the overall time complexity for this variant is $O(|S| \log |S| \log |I|)$.

- *Top-k most used nodes to start a trip during time interval $[t_1..t_2]$(`top_K`$_{Ts}$).* Following the same guidelines discussed above for `top_K`$_T$, adapting the sequential and binary-partition solutions for the spatial `top_K`$_s$ to include temporal constraints is straightforward, making its time complexity $O(|S| \log |S| \log |I| \log n)$.

4.4 Experiments

We have run experiments to evaluate both the space requirements and performance at query time of CTR when dealing with spatial, temporal, and spatio-temporal queries over two different datasets (Porto and Madrid) that are described in Section 4.4.1.

We have used several configurations of CTR by tuning both its spatial and temporal components. In the spatial part, we set the Ψ sampling parameter (t_Ψ) to the values $t_\Psi \in \{32, 128, 512\}$. For the temporal component, we have tested both the balanced WM, and the Hu-Tucker-shaped WT (HTWT) using the bitvector configurations discussed in Section 4.2.2.1. That is, using either a plain bitvector *RG* with a sparse sampling (RG_{32}), or a *RRR* bitvector with sampling parameter $\in \{32, 64, 128\}$ (RRR_{32}, RRR_{64}, and RRR_{128}).

4.4.1 Experimental datasets

We used two different datasets of trips in our experiments:

- **Madrid dataset**: Using GTFS data obtained for the public transportation network of Madrid,[12] we generated a dataset of synthetic trips combining the subway network with the Spanish commuter rail system.[13] (called *cercanías*) In this dataset, we have defined the nodes as stops or stations, making two of them connected if there exists a line or route that stops at both nodes, consecutively, thus allowing us to represent user trajectories following the same strategy as for urban street networks. In total, there are 313 different stations/nodes from 23 lines.

  We generated 10 million trips with lengths varying from 2 to 31 nodes traversed. Those lengths follow a binomial distribution. The average length of the trips is 11.81 nodes.

  In the generation of a trip of length l, we randomly choose a starting node from a line, and the starting direction. Then, we follow that line until we reach a switching node. At this node, we decide whether to follow the current line or to switch to a new line. We allow only up to four line switches for a given trip, and use fixed probability values to decide whether to switch line or not. Such probability is 0.5, 0.1, 0.05, and 0.02 respectively for the first, second, third, and fourth line switch in a trip. We also avoid revisiting nodes in the same trip. The generation process ends when l nodes have been added to the trip, or a dead end is reached.

  As a baseline, the plain representation of the generated trips using a 9-bit integer ($\lceil \log_2 314 \rceil = 9$) for every node-ID (and the \$ separator) would require 137.47 MiB, while requiring a sequential processing on the whole collection to answer most of our proposed queries.

  We also generated synthetic times for those trips following the same rules used to create the time distribution named *skewed* in Figure 4.7, so most of the trip timestamps belong to rush hours. Instead of using only regular working days, we distinguished four kinds of days in a week: regular working days; Fridays and holiday eves; Saturdays; and Sundays and holidays. We also assume that there are two kinds of weeks related to high and low season periods. Therefore, a time interval may belong to eight types of day. When discretized at five-minute intervals we obtain 2,304 distinct time intervals, while when we use thirty-minute intervals we obtain 384. In the former case, our baseline for the generated times using 12 bits per time-ID would occupy 183.30 MiB. In the latter one, each time-ID requires 9 bits and the temporal baseline requires 137.47 MiB.

[12] Data from the CRTM corporation at `http://www.crtm.es/`.
[13] Data manually scraped from public sources.

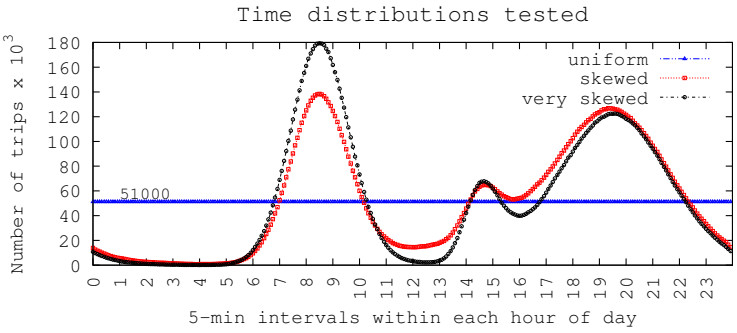


Figure 4.7: Time distributions tested. The final Madrid dataset was generated with the distribution called *skewed*. The y-axis indicates the number of passengers per each 5-min interval.

- **Porto dataset**: We downloaded a collection of 1,710,671 trajectories from the city of Porto corresponding to taxi trips during a full year (from July 1, 2013 to June 30, 2014), provided by [MMGF+13].[14] Among other fields those data include, for each taxi ride, a list of GPS coordinates and times gathered every 15 seconds of the trip. We adapted such data to our needs by using a map matching algorithm provided by the Graphhopper library,[15] and OpenStreetMap cartography.[16] With this, we could determine what streets segments were traversed by the trips from their list of coordinates. Finally, trips were encoded as a sequence of identifiers corresponding to adjacent stretches of street (that is, basic street segments with no intersections) the trip traversed, each one of them tagged with a timestamp.

  After filtering incomplete matches, 1,617,774 trips, built over 59,618 distinct street segments, were used for the dataset. Due to the nature of the network and the trips, the average number of street segments per trip is 64.74; that is, the length of the trips is longer than in Madrid dataset. Since we needed $16 = \lceil \log_2 59{,}618 \rceil$ bits to represent each segment in a trip, the total size of our plain spatial baseline is 202.85 MiB.

  For the temporal part, we considered only one kind of day. Therefore, when we sample those 24 hours into five-minute intervals, we obtain 288 distinct time intervals that are given a 9-bit time-ID. Consequently the overall size of the temporal baseline becomes 114.10 MiB. However, if we split those 24 hours into thirty-minute intervals, only 48 time intervals arise. In this case, each time-ID needs only 6 bits and the total size of the temporal baseline is 76.07 MiB.

Note that, although this is not the best representation for trips over public transportation

[14] Description at `http://www.geolink.pt/ecmlpkdd2015-challenge/dataset.html` . Downloaded from `https://archive.ics.uci.edu/ml/machine-learning-databases/00339/train.csv.zip`

[15] `https://github.com/graphhopper/map-matching`

[16] `http://www.openstreetmap.org/`

networks (see Chapter 5 for our proposed representations for those cases), we have decided to include the experiments over the Madrid dataset anyway. The reason for its presence is twofold: One, to show a realistic use case of this representation built over a public transportation network instead of a context of streets or roads, so the problems discussed in the next chapter could be understood more clearly. Two, because it serves as a better example for space-time trade-offs of our two alternative structures used to represent time intervals: the HTWT and the WM.

4.4.2 Space Requirements

We show the compression obtained by CTR when built on our two test datasets. Compression is shown as the percentage of the size of the plain baselines discussed above. Using different configurations of CTR, we will show the compression of the spatial component (CSA), that of the temporal component (HTWT and WM), and finally the overall compression of CTR.

	t_Ψ		
	32	128	512
Madrid	41.32%	26.80%	23.06%
Porto	23.66%	15.49%	13.37%

Table 4.1: Compression of CSA with respect to the spatial baseline.

Results regarding the compression obtained by CSA are given in Table 4.1. The compression ratio is calculated over a plain spatial-only (stop-IDs or street-segment-IDs in each case) representation. Note that an *iCSA* built on English text [FBN+12] typically reached the compression of *gzip* (around 35% in compression ratio). As expected, the high compressibility of our sorted datasets of trips helps our CSA to improve those numbers with compression ratios under 30%, while also offering indexing features that allow us to perform efficient searches. In a rather dense configuration of CSA with $t_\Psi = 32$ we obtain compression ratios around 41% and 23% for Madrid and Porto datasets respectively. Those results are interesting from the simple fact that the baseline representations were only using respectively 9-bits per node (Madrid) and 16-bits per segment (Porto). As expected, compression improves as we increase the Ψ sampling parameter t_Ψ. We show that by tuning CSA in a more sparse setup we can almost halve the space needs of using $t_\Psi = 32$, although the resulting CSA would become much slower as we will later prove in Section 4.4.3. In general, we can see that CSA obtains better compression in Porto than in Madrid. This is probably due to the longer and more predictable trips. Note that is not common to arrive at an intersection having more than two valid street links where to navigate to.

In Table 4.2, we focus on the space needed by the temporal component of CTR. In this case we show the compression ratios obtained by HTWT and WM considering that time is either discretized into 5-min or 30-min intervals. Recall that the size of the plain baseline representations differs depending on the discretization period. Both HTWT and WM were

	Type of bitvector in WM/HTWT			
	RG_{32}	RRR_{32}	RRR_{64}	RRR_{128}
Madrid (HTWT, 5-min)	91.33%	80.89%	76.90%	74.90%
Madrid (WM, 5-min)	103.13%	86.03%	80.61%	77.88%
Madrid (HTWT, 30-min)	92.30%	78.90%	74.66%	72.52%
Madrid (WM, 30-min)	103.14%	83.32%	77.90%	75.18%
Porto (HTWT, 5-min)	93.52%	102.61%	98.27%	96.11%
Porto (WM, 5-min)	103.13%	106.88%	101.41%	98.66%
Porto (HTWT, 30-min)	96.00%	103.78%	99.08%	96.74%
Porto (WM, 30-min)	103.12%	107.00%	101.50%	98.75%

Table 4.2: Compression of WM and HTWT with respect to the temporal baseline.

tuned by using bitvector representations RG_{32}, RRR_{32}, RRR_{64}, and RRR_{128}, as indicated above.

One important insight from these results is that in the synthetic dataset from Madrid *RRR* bitvectors always lead to a better compression than the plain *RG*, while in the real dataset from Porto that is not always the case, and we have to use the sparsest configuration of *RRR* to achieve similar space requirements of the uncompressed *RG* version. Consequently, for Porto dataset, the faster plain *RG* bitvectors are probably the best choice. In Madrid dataset, we can see an actual space/time trade-off: *RRR* obtains better compression but will be slower, as later seen in Section 4.4.3.

Finally, in Table 4.3, we show the overall compression rates of CTR. We use the same configurations for HTWT and WM as in Table 4.2, and both the most dense and sparse tuning of CSA ($t_\Psi = 32$ and $t_\Psi = 512$ respectively). For Madrid dataset, the pair (node, timestamp) is represented with $9 + 9 = 18$ bits in our baseline representation when time is discretized into 30-minute intervals, and with $9 + 12 = 21$ when we use 5-minute intervals. In the case of Porto dataset, when using 30-minute intervals, each pair (node, timestamp) from the baseline requires $16 + 9 = 25$ bits. If discretization considers 5-minute intervals, the baseline requires $16 + 6 = 22$ bits. We can see that the overall compression of CTR in Madrid dataset ranges between 76% and 50%. Also we show that Porto dataset is much more compressible, obtaining compression ratios from around 50% down to 35%.

4.4.3 Performance at query time

Through this section, we evaluate the time performance of CTR when solving spatial, temporal, and spatio-temporal queries. We have randomly generated 10,000 query patterns from our two datasets for each type of query. Each time measurement presented below is

		Type of bitvector in WM/HTWT			
		RG_{32}	RRR_{32}	RRR_{64}	RRR_{128}
$t_\Psi = 32$	Madrid (HTWT, 5-min)	69.90%	63.93%	61.65%	60.51%
	Madrid (WM, 5-min)	76.64%	66.87%	63.77%	62.21%
	Madrid (HTWT, 30-min)	66.81%	60.11%	57.99%	56.92%
	Madrid (WM, 30-min)	72.23%	62.32%	59.61%	58.25%
	Porto (HTWT, 5-min)	48.81%	52.08%	50.52%	49.74%
	Porto (WM, 5-min)	52.27%	53.62%	51.65%	50.66%
	Porto (HTWT, 30-min)	43.39%	45.51%	44.23%	43.59%
	Porto (WM, 30-min)	45.33%	46.39%	44.89%	44.14%
$t_\Psi = 512$	Madrid (HTWT, 5-min)	62.07%	56.10%	53.82%	52.68%
	Madrid (WM, 5-min)	68.81%	59.04%	55.94%	54.38%
	Madrid (HTWT, 30-min)	57.68%	50.98%	48.86%	47.79%
	Madrid (WM, 30-min)	63.10%	53.19%	50.48%	49.12%
	Porto (HTWT, 5-min)	42.22%	45.49%	43.93%	43.15%
	Porto (WM, 5-min)	45.68%	47.03%	45.06%	44.07%
	Porto (HTWT, 30-min)	35.91%	38.03%	36.75%	36.11%
	Porto (WM, 30-min)	37.85%	38.91%	37.41%	36.66%

Table 4.3: Overall compression of CTR including different configurations for both the spatial and temporal components.

the average execution time of 10,000 runs using the corresponding query patterns, except for the `top_K` queries where we perform 100 runs of the *top-k* algorithms with $k \in \{10, 100\}$.

Our test machine has an Intel(R) Core(tm) i5-4690@3.50GHz CPU (4 cores/4 siblings) and 8GB of DDR3 RAM. It runs Ubuntu Linux 16.04 (Kernel 4.4.0-21-generic). The compiler used was GCC version 5.4.0 and we set compiler optimization flags to $-O3$. All our experiments run in a single core and time measures refer to CPU user-time.

During the generation of query patterns, for those queries involving only one node X from the network, we have randomly chosen X 10,000 times from the available network nodes. This is the case of the query patters used both for the spatial queries `start_X`, `end_X`, and `load_X` or the spatio-temporal `start_X`$_T$, `end_X`$_T$, and `load_X`$_T$. In the case of the spatial `from_X_to_Y` and the spatio-temporal `from_X_to_Y`$_{Ts}$, and `from_X_to_Y`$_{Tw}$ the pair of network nodes $\langle X, Y\rangle$ that compose our query patterns were generated by randomly choosing 10,000 trips and then extracting the initial X and ending Y nodes of those trips, in order to avoid generating queries for pairs $\langle X, Y\rangle$ that were absent in our dataset.

Moreover, we also generated the time intervals $[t_1..t_2]$ required for the spatio-temporal

queries. Considering the different available time-IDs, we chose a random starting instant t_1 and then randomly generated the width of that interval from five minutes to two hours. Note that if we discretized time into 5-minute intervals and *interval-width* = 59 minutes, our time interval $[t_1..t_2]$ would contain exactly 12 time IDs ($t_2 \leftarrow t_1 + 11$). However, if time was discretized into 30-minute intervals, $[t_1..t_2]$ would contain only 2 time IDs ($t_2 \leftarrow t_1 + 1$). We followed the same procedure to generate the query patterns used for the pure temporal queries `load_T` and `start_T`.

4.4.3.1 Space/time trade-off when dealing with spatial queries

In Figures 4.8 and 4.9, we show the performance of CTR at solving spatial queries for Madrid and Porto datasets respectively. Note that all these queries can be answered using only the CSA component of CTR. Therefore, the size of the temporal component is not considered here and compression values (x-axis) refer only to the size of CSA with respect to the spatial baseline as in Table 4.1. We show the average query time (in μs) depending on the space used by CSA with three different sampling configurations ($t_\Psi \in \{512, 128, 32\}$).

Results show that the queries that involve searching in the \$ region of Ψ, such as `start_X` or `from_X_to_Y` are considerably slower than queries `end_X` and `load_X` due to the large size of that region when compared to the frequency of any node: in neither of the two datasets there is a node that was visited by every trip, while there is one \$ per trip.

In both datasets, we can see that `load_X` (solved using select on D rather than bsearch on Ψ) is the fastest query. On average, it takes only around 10ns per query. Except in the most sparse configuration of CSA, queries `end_X`, `start_X`, and `from_X_to_Y` require typically less than 10μs. This basically shows the cost of performing bsearch on a compressed Ψ. In the most sparse setup ($t_\Psi = 512$), times for `start_X` and `from_X_to_Y` are always better for the dataset of Madrid than for the one of Porto, and `end_X` draws rather identical times. With the densest configuration ($t_\Psi = 32$), `end_X` and `from_X_to_Y` are respectively around 10-20% fastest in Madrid dataset (`end_X` takes 4.05μs and 4.51μs respectively, and `from_X_to_Y` takes 4.54μs and 5.66μs). However, `start_X` performs around 20% faster in Porto dataset (2.28μs vs 2.90μs).

Focusing on `top_K` queries, we can see huge differences between `top_K`$_s$ and the rest of the `top_K` queries, as the former needs to perform bsearch over the compressed Ψ instead of a select on D.

We can also see that due to the small number of stops in Madrid dataset, it is always more efficient to use the sequential version of `top_K`$_s$ and `top_K` algorithms. This is also because a rather uniform frequency among nodes increases the number of insertions in the priority queue (i) of the binary-partition algorithm needed for retrieving the first k nodes ($i \approx |S|$). Moreover, note that for the sequential algorithm i is at most $|S|$, whereas for the binary-partition counterpart it could become up to $2|S| - 1$.

However, in Porto dataset, where nodes follow a biased distribution (some streets are much more used than others by taxis), and a vocabulary is 190 times larger than the one for Madrid, the binary-partition version of `top_K`$_s$ and `top_K` algorithms is clearly faster than the sequential counterpart (`top_K`$_{seq}$ and `top_K`$_{sseq}$). Note that in Madrid dataset, *top*_100 returns 32% of the nodes (hence sequential processing worths it) whereas in Porto dataset less than 0.2% of the nodes are returned.

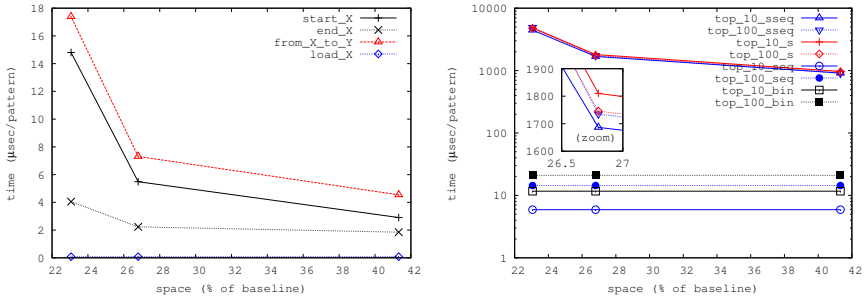


Figure 4.8: Spatial queries (left) and spatial *top-k* queries (right) for Madrid.

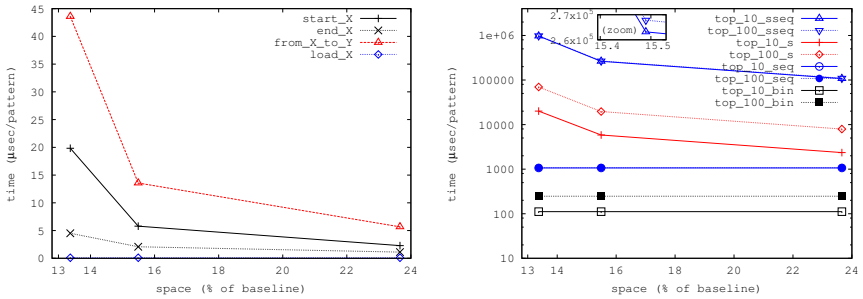


Figure 4.9: Spatial queries (left) and spatial *top-k* queries (right) for Porto.

The gap between top_10_{seq} and top_100_{seq} that we can clearly appreciate in Madrid dataset is due to the cost of the insertion of nodes in the min-heap. However, the gap between the binary top_10_{bin} and top_100_{bin} is mainly related to the number of iterations performed until the binary-partition algorithm gathers the first 10 and 100 nodes returned respectively. The same discussion applies for $\texttt{top\_K}_s$ queries.

4.4.3.2 Comparing the space/time trade-off of WM and HTWT

In order to compare the efficiency of our HTWT (that uses variable-length codes and supports count efficiently) with a balanced WM alternative under different time distributions (recall that this WM is time distribution invariant), we run some experiments that evaluate the average time to execute count operation on both representations.

We used a dataset of generated trips for Madrid (refer to Section 4.4.1 for details) and we generated three kinds of time distributions for our evaluation. We refer to them as: uniform, skewed, and very skewed, as they are shown in Figure 4.7. According to the total number of passengers in a day, in the uniform distribution, 51,000 passengers use the network for each 5-minute interval. We also generated a skewed distribution for the

time interval frequencies in an effort to model the usage of a public transportation network in a regular working day, where the starting time of a trip is generated according to the following rules:

- With 30% of probability, a trip occurs during a morning rush hour.
- With 45% of probability, a trip occurs in an evening rush hour.
- With 5% of probability, a trip occurs during lunch rush hour.
- The remaining 20% of probability is associated to unclassified trips, starting at a random hour of the day, which may also fall into one of the three previous periods discussed.

In the very skewed distribution we increase the rush-hour probabilities with 40% for the morning rush hour, 50% for the evening rush hour, 8% for lunch period and only 2% of random movements.

For these generated datasets, we have built the HTWT and the WM considering two different granularities for the discretization of times: five-minute and thirty-minute intervals. Then, we generated 10,000 random intervals of times $[t_1..t_2]$ over the whole time sequence of the dataset considering interval widths of five minutes, one hour, and six hours. Finally, we run 10,000 $\text{count}_{t_1,t_2}(Icode^{\Psi}, |\mathcal{T}|+2, n)$ queries (we show average times) from each query set over the six configurations of HTWT and WM (2 different granularities for the time discretization and 3 datasets).

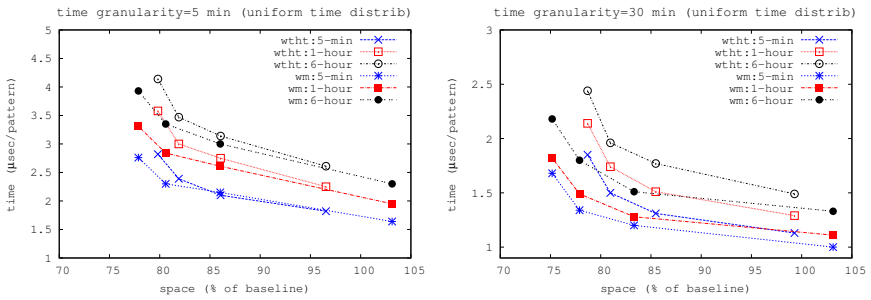


Figure 4.10: Space/time trade-offs for *count* queries with a uniform the time distribution. Time granularity for the time index is 5 minutes (left) or 30 minutes (right).

In Figures 4.10,4.11 and 4.12, we show the results of our experiments. In Figure 4.10, we include the results for HTWT and WM built over the times assuming uniform frequency distribution. In Figure 4.11 we assume times follow a the skewed distribution, and in Figure 4.12 we show results when considering a very skewed distribution. Moreover, plots in the left column show results for our structures considering that a 5-minute granularity is chosen for the discretization of times, whereas plots on the right column assume time granularity is 30 minutes. For each scenario we include plots `wtht:5-min`, `wtht:1-hour`,

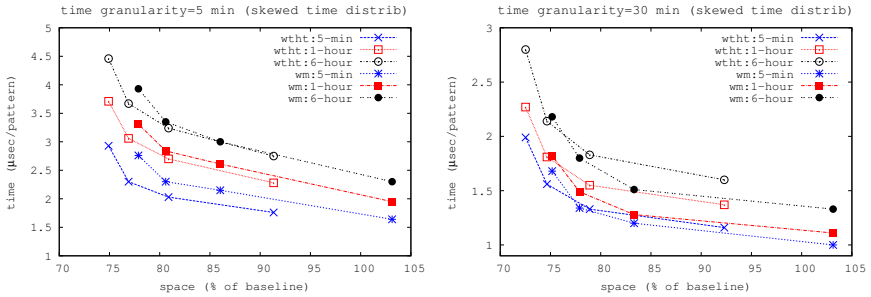


Figure 4.11: Space/time trade-offs for *count* queries with a skewed the time distribution. Time granularity for the time index is 5 minutes (left) or 30 minutes (right).

and `wtht:6-hour` for HTWT (range width for count is respectively 5-minutes, 1-hour, and 6-hours). We also present those plots for WM (`wm:5-min`, `wm:1-hour`, and `wm:6-hour`).

The baseline used for the space usage (x-axis) is the size of an array of fixed-length time-interval IDs represented with the least number of bits needed (12 bits and 9 bits respectively for 5-minute and 30-minute granularity, see Section 4.4.1).

When times are uniformly distributed, our HTWT can only exploit the redundancy introduced by the \$ symbols. With this, HTWT can obtain only a minimal compression (around $96-98\%$ of the baseline) when using a *RG* (plain) bitvector, whereas WM uses more space than the baseline (around 104%). Recall that, for each plot, we present four points corresponding (left-to-right) to RRR_{128}, RRR_{64}, RRR_{32}, and *RG* bitvectors. When using compressed bitvectors (*RRR*), WM becomes the best choice. It is both more compact (bitvectors in WM are more compressible) and faster than HTWT. In any case, using *RRR* clearly slows down queries.

A skewed distribution favors the compression for a statistical coder like Hu-Tucker, which explains the higher compression obtained. However, it also slightly increases the query times, especially in the wider one-hour and six-hours query sets. This happens because the probability of having a query that forces to descend completely up to the leaves of the HTWT also increases.

For a very skewed distribution, the gap in compression between HTWT and WM increases clearly (around 5 percentage points), whereas query times remain similar to those in the previous scenario.

As a conclusion of the experiments discussed in this section, we have shown that the distribution of the sequence of times can be exploited by our HTWT to achieve a better compression and even improved query times than the balanced WM counterpart.

4.4.3.3 Space/time trade-off when dealing with temporal queries

In this section we focus on the performance of the temporal component of CTR. We use the same configurations as in Table 4.2 for WM and HTWT (although we are only

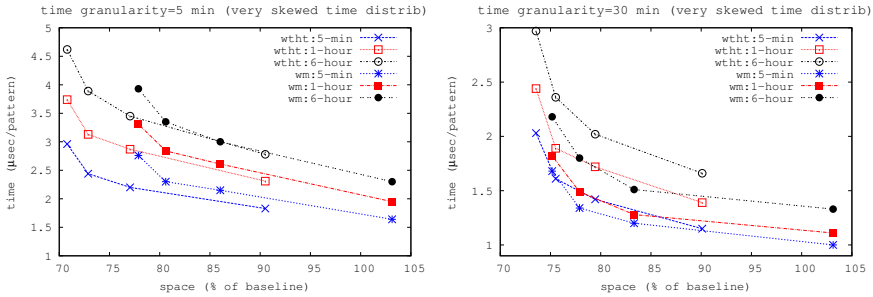


Figure 4.12: Space/time trade-offs for *count* queries with a very skewed the time distribution. Time granularity for the time index is 5 minutes (left) or 30 minutes (right).

using generated times for Madrid following the *skewed* variant, as previously stated in Section 4.4.1), and show the space/time trade-offs obtained when solving pure temporal queries. Figures 4.13 and 4.14 present the results obtained at `load_T` and `start_T` queries for Madrid and Porto datasets respectively. Note that, in this case, since the CSA is not actually needed to solve temporal queries, we do not include its size within the compression values (x-axis).

We can see that when running `load_T` queries, both HTWT and WM obtain rather similar times (requiring less than 4μs to perform a count operation in all cases) and that those times improve as the height of the structure decreases. We can see that in the highest HTWT and WM, corresponding to using 5-min intervals in Madrid dataset, `load_T` requires less than 3.5μs. Then, when using 30-min intervals, the time required to solve `load_T` is always below 2.3μs (yet WM performs faster than HTWT here), and those times are similar to the ones obtained for Porto dataset when using 5-min intervals. Finally, the best query times (below 1.2μs) are obtained for Porto dataset with 30-min intervals.

Regarding `start_T`, recall that it also performs a count operation, but within a smaller range ($[2..|\mathcal{T}+1|]$) in comparison with the range $[|\mathcal{T}|+2..n]$ where count is performed for `load_T`. We can see that, whereas the WM obtains similar times to those of `load_T` query, `start_T` performs clearly faster than `load_T` over the HTWT.

While query times are bounded by $O(\log |I|)$, as seen in Section 4.3.2, it is observed that, in some cases, `start_T` is answered considerably faster than `load_T`. This occurs because, for some of our randomly queried time intervals (of up to two hours), there are no trips starting around that queried time, allowing the count operation to be cut short before reaching the leaves of the HTWT or the last level of the WM. The difference is obviously accentuated in the case of HTWT, as the tree is not uniform.

As a final note, recall that in Madrid dataset, bitvector *RG* always needs more space than *RRR* counterparts whereas in Porto dataset (as discussed in Section 4.4.2) *RG* obtains the best space values when using 5-min intervals and still requires less space than RRR_{32} when using 30-min intervals. This is the reason why while plots for Madrid dataset are decreasing from left to right, in Porto dataset the first point (*RG*) in the left figures (5-min

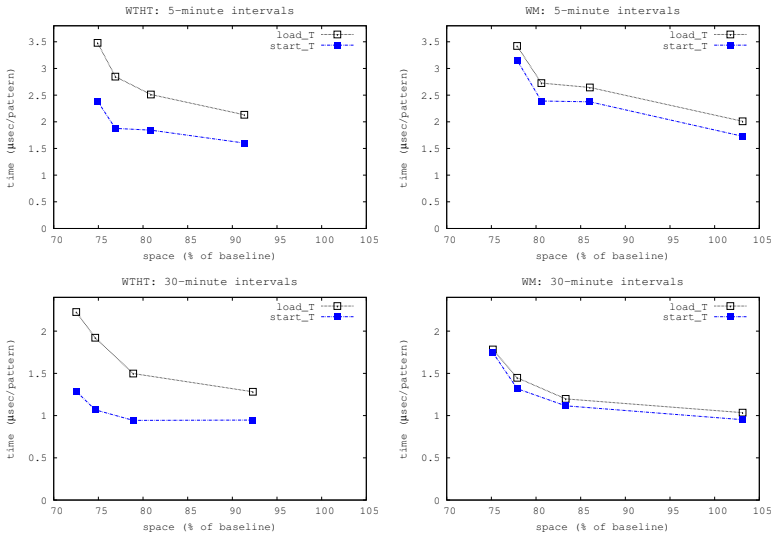


Figure 4.13: Pure temporal queries for Madrid, using either a HTWT (left) or a WM (right). Time granularity is 5 minutes (top) or 30 minutes (bottom).

intervals), and the third point (*RG*) in the right figures (30-min intervals) require less space than the others (*RRR*) and are also typically faster.

4.4.3.4 Space/time trade-off when dealing with spatio-temporal queries

In Figures 4.15 and 4.16, we show the space/time tradeoff obtained by CTR when dealing with spatio-temporal queries. Recall that this type of queries require both using the CSA, to exploit indexed access to the nodes in the trips, and the temporal component of CTR to handle temporal constraints. In this case, the space values showed in the figures include both the size of CSA and that of either WM or HTWT. Therefore, we also show the overall space needs of CTR. In the case of CSA we have set $t_\Psi = 32$ (a fixed dense sampling), and for WM and HTWT we used again the same configurations as in the previous sections obtained by varying the bitvectors and the temporal discretization.

For queries $\texttt{start\_X}_T$, $\texttt{end\_X}_T$, and $\texttt{load\_X}_T$ we can see typically small differences between using WM or HTWT. In Madrid dataset, WM overcomes HTWT being 2-30% faster in these types of queries. However, in Porto dataset HTWT is slightly faster (from 1 to 25%) than its WM counterpart.

For queries $\texttt{from\_X\_to\_Y}_{Ts}$ and $\texttt{from\_X\_to\_Y}_{Tw}$ we can see a big gap between the times reported by HTWT and WM. This gap arises because in WM we have used the count^{LR} operation discussed in Section 4.3.3 that was implemented with two additional upward traversals for the WM.[17] However, in our implementation of HTWT we have engineered

[17]We used the exact same WM implementation as in [CNO15] and simply added the new

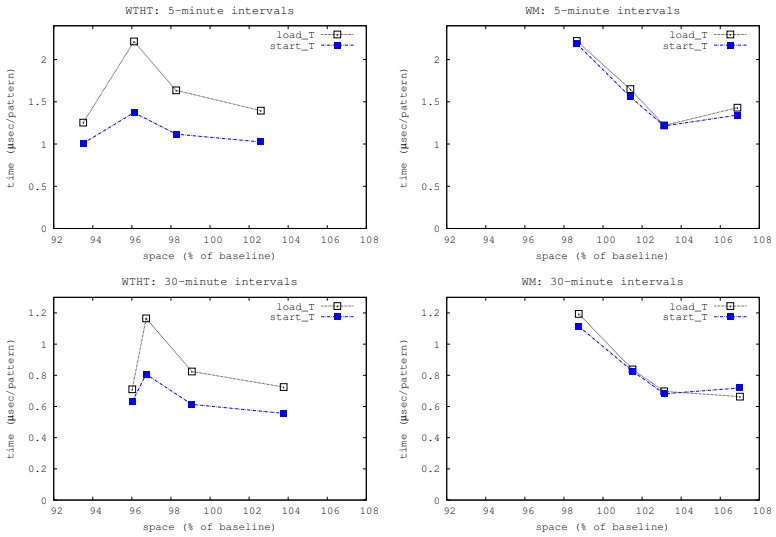


Figure 4.14: Pure temporal queries for Porto, using either a HTWT (left) or a WM (right). Time granularity is 5 minutes (top) or 30 minutes (bottom).

an improved version of count^{LR} where, during the execution of count, we also report α' and β', hence avoiding extra operations.

Finally, we also include results for `top_K`$_T$ and `top_K`$_{Ts}$ queries in Figures 4.17 and 4.18. As explained in Section 4.4.3.1, the sequential approach is preferred when the frequency distribution of nodes is rather uniform (Madrid dataset). Otherwise, the binary-partition counterpart outperforms it. The need for applying a temporal constraint simply accentuates this effect in comparison with the corresponding pure spatial queries.

operation count^{LR} that calls the underlying count from the WM and traverses up from the bottom level by using the select operation over its bitvectors. In later works, we have optimized this operation.

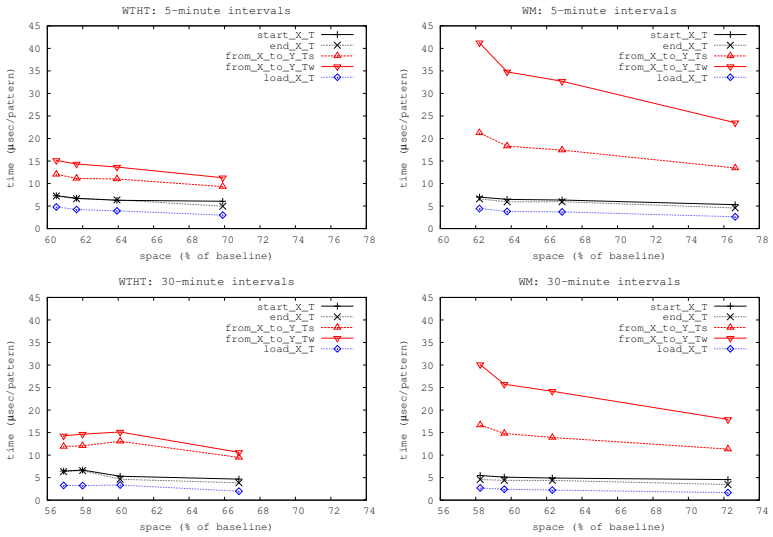


Figure 4.15: Spatio-temporal queries for Madrid, using either a HTWT (left) or a WM (right). Time granularity is 5 (top) or 30 minutes (bottom).

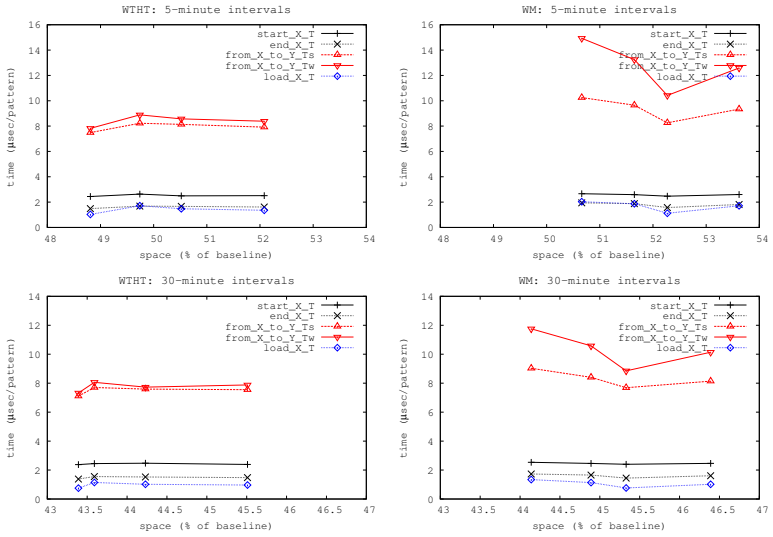


Figure 4.16: Spatio-temporal queries for Porto, using either a HTWT (left) or a WM (right). Time granularity is 5 (top) or 30 minutes (bottom).

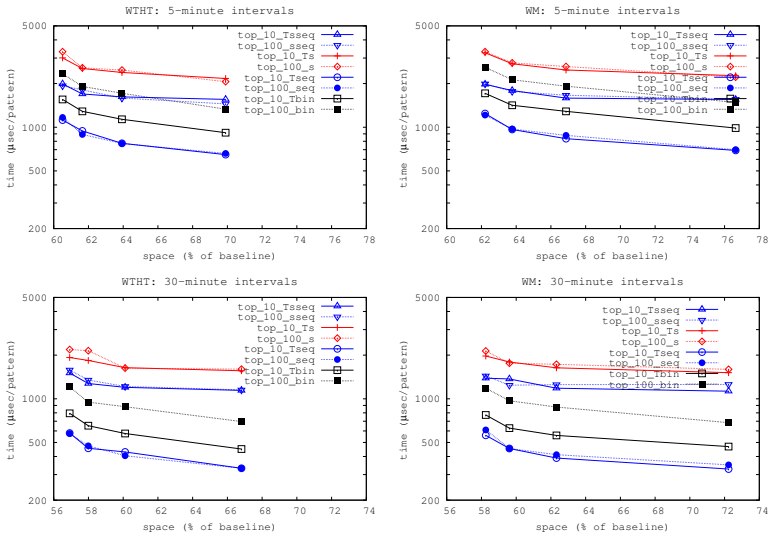


Figure 4.17: Spatio-temporal `top_K` queries for Madrid, using a HTWT (left) or a WM (right). Time granularity is 5 (top) or 30 minutes (bottom).

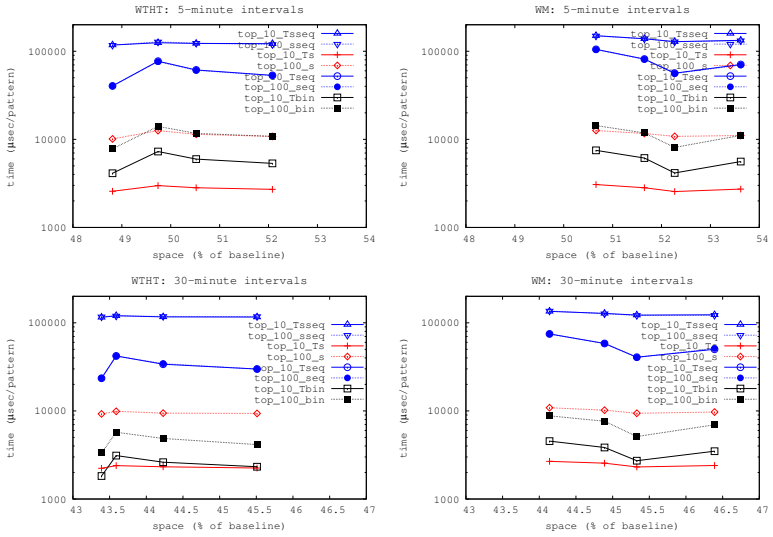


Figure 4.18: Spatio-temporal `top_K` queries for Porto, using a HTWT (left) or a WM (right). Time granularity is 5 (top) or 30 minutes (bottom).

Chapter 5

Representations for trips over public transportation networks

After CTR was introduced in Chapter 4 as a representation oriented for trips over urban street networks, in this chapter we introduce two alternative representations, called Topology&Trip-aware Compact Trip Representation (TTCTR) and eXtended Compact Trip Representation (XCTR). These new techniques are expected to be more adequate for trips over public transportation networks, since they make it possible to query about network concepts such as line and schedules, and also use them in order to reduce the size of our structures.

Both TTCTR and XCTR are also based on the CSA and the WM, although they rely on other common structures for a network representation, that do not need to be compact due to their already small size. However, we have found out that while these representations excel at many of our proposed queries related to trip patterns, they could be rather inefficient for other kinds of aggregation queries about the load of a network. For this reason, we have also introduced a complementary Trip-Matrices (T-Matrices), a SAT-based structure designed to accelerate this second kind of queries.

We have implemented and evaluated our structures, over the current bus network of Madrid, thus analyzing, both theoretically and experimentally, the fitness of each representation for every use case. This chapter starts with a brief description of the scenario where we identify all the elements involved in a public transportation network, and we also identify the most significative queries that should be considered. Then, in Section 5.2, we present all the structures considered in our proposal. This includes the common structures to manage network-related elements, and the structures TTCTR, XCTR, and Trip-Matrices (T-Matrices). In Section 5.3, we show how to support queries. Finally, we analyze them theoretically in Section 5.3.3, and experimentally in Section 5.4, where we measure both the space needs and query performance. Additionally, in Chapter 7 we will show how these representations can be integrated into a GIS application, usable for a transportation

network administrator.

5.1 Description

While, in theory, we could use the CTR (Chapter 4) to also represent trips over a subway or bus network, such representation would be very redundant. If we defined our CTR nodes as stops or stations, making two of them connected if there exists a line or route that stops at each, we would find out that a commuter would board at a stop and follow only one route (line), passing through all its stops consecutively until the alighting stop, either to switch lines or end the trip. Furthermore, a single vehicle (such as a bus or a train) will be shared by several commuters at the same time, thus also producing trips that visit the same nodes at the same times. Because simply listing every node traversed introduces all these kinds of redundancy for public transportation networks, we state that CTR is more adequate for urban street networks.

Therefore, in order to better capture the information regarding trips over a public transportation network, and to exploit this information in order to find a representation that permitted us to reduce redundancy, we proposed an ER model that handles all the information related to the demand on a public transportation network. It is shown in Figure 5.1.

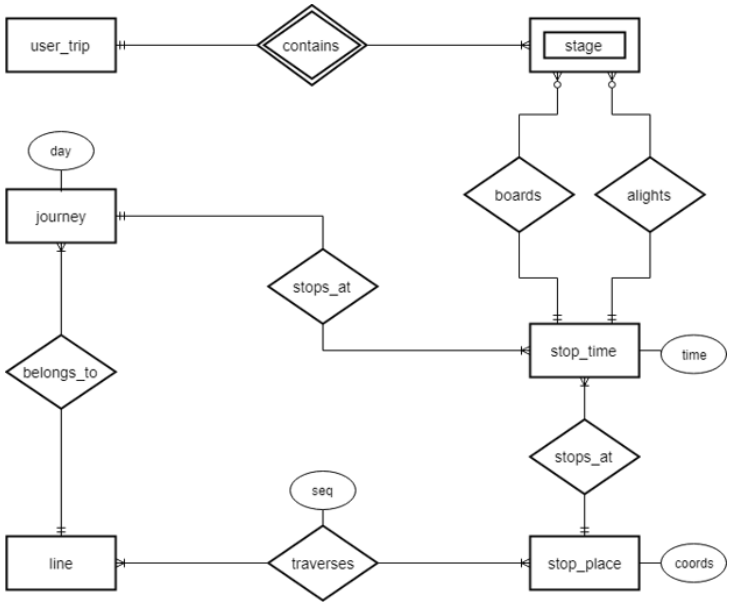


Figure 5.1: An ER diagram representing our model of user trips for public transportation networks.

These are the main elements from our model:

- A **stop_place** is a physical stop with a location, on which several lines may make stops.
- A **line** is an ordered sequence of stop places that can be traveled by a transport vehicle, such as a bus or a train. It only considers one travel direction. For this reason, there is often a different and complementary line for the opposite direction.
- A **journey** is a singular traversal of a transport vehicle over a line. It can be seen as a vehicle trip, instead of a user trip.
- A **stage** is formed by a boarding from a stop and an alighting to another from the same single line and journey.
- An **user_trip** is a concatenation of several stages, until the final destination (alighting stop of the last stage) is reached.

This approach allows us to treat the information in a layered fashion: the bottom layer is a static network representation, formed by the line and stop_place types, the middle layer represents the journeys made by vehicles that make stops at specific times, while the top layer includes the trips made by the users in these vehicle journeys. Finally, it is possible to introduce a **user** identity, with an anonymized identifier to split trips by users. However, we have not considered such information useful for the kind of analysis that this work focuses on. If needed in the future, this additional entity could be trivially integrated in our representation.

In order to represent and operate our data structures, we will follow this model by defining stops $s_i \in S$, lines $l_i \in L$ and journeys $j_i \in J^l$. It is important to state that journeys are **not** identified by j_i, as the same j_i can belong to several J^l from different lines, so we speak about journey **codes** (jcodes) instead of journey identifiers. In Figure 5.2 we can find an example network with two lines and fourteen stops, and journeys that periodically traverse these lines. Note that journey-code 0 appears both in line 0 and in line 1.

A user trip can be represented by the stops from the transportation system that were boarded by a user, so from now on we will consider a trip as a sequence of triplets $<s, l, j>$, where s and l are, respectively, stop and line identifiers, while j are the journey codes corresponding to the journeys that compose the trip. These triplets describe a trip in a consecutive fashion, on the same order as the stops were boarded. Additionally, as we are interested in knowing where the trips end, we also represent the last stop where the user has alighted. Note that both its line and journey will logically match the line and journey of the last boarding stop. Although it is generally hard to obtain information about the last destination stop of a trip, many transportation companies are investing effort in providing it, either by implementing systems to keep track of users as they leave their system or estimating it based on previous trips made by that user [AAMF16].

Example 5.1: The arrows in Figure 5.2 are examples of five user trips done along the network. For example, there is a user trip (dashed arrow from $S3$) that starts at stop $S3$ at 06:25 on *day-1*, following the journey 1 of line 1 until $S10$, where the user switches to line 2 at time 06:35 and continues along the journey 2 of line 2 (the one started as 06:30 in $S13$) up to stop $S12$. Consequently, this trip includes two stages. □

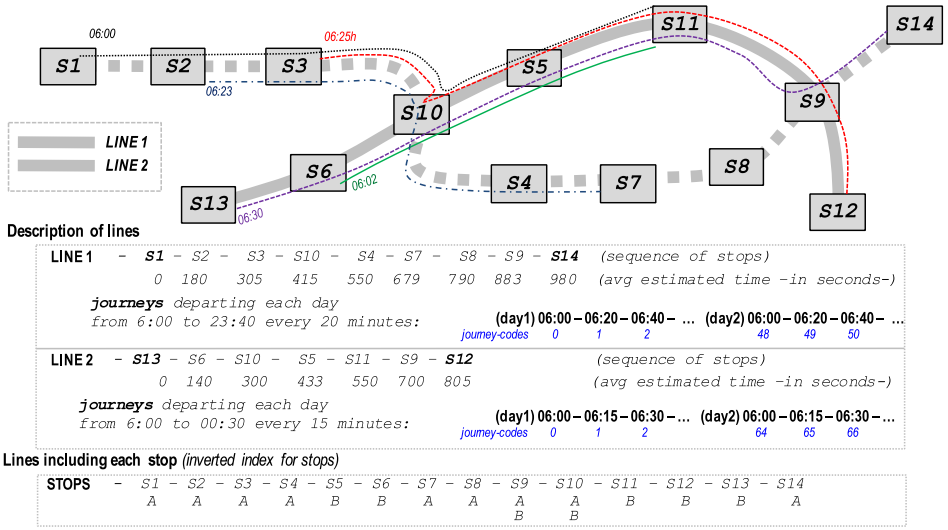


Figure 5.2: Network representation with the common structures.

In our first representation, TTCTR, we encode all valid $<s, l>$ pairs into a vocabulary V, with every trip defined as a concatenation of the $<s, l>$ pairs for the boarded stops, ended by the final destination stop, which will be alighting. We build a CSA over the concatenation of these trips, and a parallel WM with the journey codes of the boarded stops, as previously done for CTR in Chapter 4, but with the temporal component represented by line-dependant journey codes instead of explicit time intervals. For our alternative representation, XCTR, we only encode the sequence of boarded stops, while the lines go into a parallel WM, and the journey codes are in a second WM that is aligned to the last level of the first one, allowing us to have more flexibility for some queries, while sacrificing efficiency in others. Finally, T-Matrices represents a matrix M_l^b for each line l where each cell in the matrix stores the number of boardings (or alightings) performed in the s-th stop of the j-th journey of line l.

In the context of public transportation networks, we are interested in solving two main kinds of queries, which we present with a non-comprehensive list of examples that can be solved with the structures proposed in this work:

A) Queries about the **network load**, asking for the gross number of users that boarded or alighted within a stop and a given time/journey. Furthermore, it can be also interesting to obtain the average load of a bus or a train between any two stops from its line. Some of those queries are:

- `board_X`$_{LT}$. Number of users that boarded a vehicle at stop X, optionally restricting to a line L and a time range T.
- `alight_X`$_{LT}$. Number of users that alighted a vehicle at stop X, optionally restricting to a line L and a time range T.

- `use_L`$_T$. Number of users (boarding any vehicle) for the line L, optionally restricting to a time range T.
- `board_T`. Number of users boarding (any vehicle) within a time range T.
- `alight_T`. Number of users alighting (any vehicle) within a time range T.
- `load_X`$_{LT}$. Average number of passengers traveling from the stop X to its next stop in the line L within the time range T. It can also be seen as the average load of the vehicle.

B) Queries about user **trips patterns**. With his kind of queries we can obtain the number of times a stop was used to switch lines or the number of trips that started on a stop with another specific stop as the final destination. In this work we consider the following queries of this kind:

- `start_X`$_{LT}$. Number of user trips starting at a given stop X, optionally restricting to a line L and a time range T.
- `end_X`$_{LT}$. Number of user trips ending at a given stop X, optionally restricting to a line L and a time range T.
- `switch_X`$_{LT}$. Number of trips in which the stop X was used to switch lines, optionally restricting to a destination line L and a time range T.
- `from_X`$_{LT}$`_to_Y`$_{LT}$. Number of user trips that originate at stop X and end at stop Y, both being optionally restricted to a line and time range. A fundamental difference with the similar definition of `from_X_to_Y`$_T$ from Section 4.1 is that we no longer restrict the whole trip to a single time range, but we query for separate time filters for the starting stop X and the ending stop Y.
- `start_L`$_T$. Number of user trips starting at any stop from a given line L, optionally restricting to a time range T.
- `end_L`$_T$. Number of user trips ending at any stop from a given line L, optionally restricting to a time range T.
- `start_T`. Number of user trips starting within a time range T.
- `end_T`. Number of user trips ending within a time range T.

5.2 Structures

To the best of our knowledge, there is no indexing structure that would allow us to efficiently represent trips that could also support all the kinds of queries described in the previous section. For this reason, we propose a new solution that relies on two data structures, T-Matrices and TTCTR. The former is targeted for queries of type A, solving most aggregation queries in constant time, while the latter can be used for queries of type B. Finally, we introduce a more versatile alternative to TTCTR that we call XCTR.

5.2.1 Common Data Structures

Considering our network formed by stops $s_i \in S$, lines $l_i \in L$, and journeys $j_i \in J^l$, the following structures represent these elements. All our following representations will rely on them.

- $lineStop_i(j)$ is the j-th stop of line l_i.
- $stopLine_i(j)$ is the j-th line that makes a stop at the stop s_i.
- $avgTime_i(j)$ is the average time in seconds that it takes for a vehicle of line l_i to reach its j-th stop from the start of a journey.
- $initialTime_i(k)$ is the starting time of the journey j_k for line l_i.

With the exception of *initialTime*, all these structures are considered small enough to be represented using plain fixed-length integer arrays. In the case of *initialTime*, its size naturally grows with the amount of trips that are indexed, thus there is a motivation to reduce its size, which can be easily achieved with any technique that works on posting lists or sequences of strictly increasing numbers, many of which have been discussed and benchmarked in [CFMPN16] and [FMPC+19]. In our work we have used a simplified Vbyte+ANS compression described in [MP17] using the Zstd library.[1] In order to facilitate searches and random access, we introduced fixed-length samples on configurable intervals.

Examples of these sequences may be found in Figure 5.2, where we indicate the sequence of stops for each line (*lineStop*), the average estimated times (*avgTime*), the initial times of each journey (*initialTime*), and finally the inverted lists of lines per stop (*stopLine*).

5.2.2 TTCTR

Topology&Trip-aware Compact Trip Representation (TTCTR) was introduced in [BFG+18], as a way of representing trips that are sequences of triplets $<s, l, j>$ for every boarded stop s, with its line l, and journey j. Finally, there is also an additional triplet for the last alighted stop, which is considered to be the final destination of the trip. In TTCTR, the spatial component (the pairs $<s, l>$ for the stops and lines of a trip) is represented with a CSA where each valid pair $<s, l>$ is encoded as an integer id in the input sequence $T[1..n]$ that is used to build the CSA.

In order to build the TTCTR structure, all trips must be first sorted. If we consider that a trip is composed by m of the $<s_i, l_i, j_i>$, $1 \leq i \leq m$ triplets previously described, where the first triplet corresponds to the first boarded stop and the last triplet corresponds to the last alighted stop (final destination), then the collection of trips is sorted by the key $<s_1, s_m, l_1, j_1>$. That is, trips are initially sorted by the first boarded stop identifier. If these are equal, they are then sorted by their last stop identifier, analogously followed by the line identifier, and journey code of the first stop.

Example 5.2: By our defined criteria, the correct way of sorting the five trips depicted as arrows in Figure 5.2 is: $t_1 = \langle(1,1,0),(10,2,1),(11,2,1)\rangle$, $t_2 = \langle(2,1,1),(7,1,1)\rangle$, $t_3 = \langle(3,1,1),(10,2,2),(12,2,2)\rangle$, $t_4 = \langle(6,2,0),(11,2,0)\rangle$, and $t_5 = \langle(13,2,2),(9,1,2),(14,1,2)\rangle$.

Note that, for example, $(13, 2, 2)$ from t_5 indicates that, at stop 13, the user boarded vehicle from line 2, that corresponds to the 3-rd journey (as the first journey-code is zero), which started at 06:30. Naturally, the line and journey *ids* of the last triple of each trip are identical to the ones in the previous triple, as the commuter had to board into that line and journey before alighting from it. □

[1] https://github.com/facebook/zstd

We also need a bijective function to encode the pairs $<s, l>$. Consider a vocabulary V such that:

- Entry $V[0]$ is reserved for the terminator symbol \$.
- Entries $\langle V[1], V[2], \ldots V[|S|])]$ are associated with stops $s_1, s_2, \ldots, s_{|S|}$ and are used to represent the final stops of the trips. That is, when a given stop s_i ends a user trip, it is given $id \leftarrow s_i$.
- The following $|L| \cdot |S|$ entries are associated with the sequence composed of the pairs $<s, l > \in S \times L$, sorted first by the stop id s and later by the line id l. That is, entry $V[|S| + 1]$ is given to $<s_1, l_1>$; $V[|S| + 2]$ to $<s_1, l_2>$; $V[|S| + 3]$ to $<s_1, l_3>$; $\ldots$; $V[|S| + |L|]$ to $<s_1, l_{|L|}>$; $V[|S| + |L| + 1]$ to $<s_2, l_1>$, $V[|S| + |L| + 2]$ to $<s_2, l_2>$, and so on. Therefore, it is easy to see that any $<s_i, l_j>$ is going to be associated with the entry $V[|S| + |L|(i - 1) + j]$.

This arrangement would theoretically produce many entries in V that are mapped to pairs $<s, l>$ that are unused in T, either because the stop is never traversed by that line or (rather unlikely) because we do not have the record of a user trip containing it. These unused entries can be skipped with a compact bitvector B with rank and select capabilities, that marks with a one every used entry from V. This will enable us to operate with a much smaller vocabulary V' with only the used entries from V, such that $V[i] = V'[\text{rank}_1(B, i)]$. Refer to the vocabulary shown in Figure 5.3(2) for an example where pairs (i.e. $<s, l>$) are encoded to 43 unique identifiers in V. After that, B marks which of the entries of V actually appear in the original sequence. Finally. V' will contain only 12 entries, for each bit set to 1 in B. Note that neither V nor V' are explicitly represented in practice, as rank_1 and select_1 operations over B are enough to map and unmap, respectively, vocabulary identifiers.

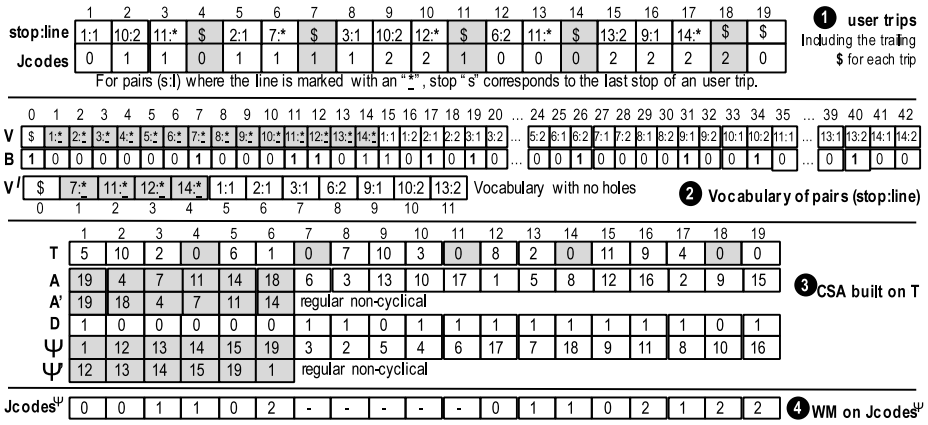


Figure 5.3: Structures involved in the creation of a TTCTR.

After this, the sequence $T[1..n]$ is built, with the identifiers obtained by mapping the pairs $<s, l>$ from the user-trips to the vocabulary entries of V'. Then a CSA is built over it,

as seen in Figure 5.3(3). Each encoded trip in T is terminated with an additional \$ symbol. Even though in the final CSA we assign all these \$ a lexicographical value of 0 ($V[0]$), we assign them different correlative values during the construction of the suffix array (A) to ensure that the entries for \$ in A maintain the same order as in the original text. Finally, we make a modification on Ψ to make the entries of each \$ point to the start of its own trip instead of the next one (cyclical as in CTR, see Section 4.2). These two modifications are proven necessary for our implemented queries, at the expense of losing some of the properties of a classic CSA that are not necessary here. For reference, in Figure 5.3 we also present A' and Ψ', that show how our modifications compare to the original CSA.

The journey codes ($jcodes$) are encoded in $Jcodes^{\Psi}[1..n]$, as shown in Figure 5.3(4), that is aligned to Ψ instead of T. $Jcodes[8] = 1$ corresponds to $Jcodes^{\Psi}[14] = 1$, since $A[14] = 8$; $Jcodes[9] = 2$ corresponds to $Jcodes^{\Psi}[18] = 2$, since $A[18] = 9$; and so on. Recall that $jcodes$ are relative to their line identifiers, leading us to skip the $jcodes$ that would be aligned to the entries of Ψ belonging to the final stops (represented as "$s:*$" in V), as they lack line identifiers, which are in turn needed to identify a journey. Additionally, for the the first positions of $Jcodes^{\Psi}$, aligned with the \$ entries, we duplicate the same $jcodes$ as in the beginning of each trip.

Finally, $Jcodes^{\Psi}$ is represented with a WM. This is exactly the same strategy as in the case of the temporal component of CTR, although in this case we are encoding jcodes instead of time intervals.

5.2.3 XCTR

For certain queries, TTCTR can be rather inefficient, as we will discuss later in Section 5.3. These use cases motivated us to develop a second version, XCTR, that instead of encoding the lines into the CSA vocabulary (V), uses a second WM with the sequence of line identifiers, thus reducing the complexity of the queries that restrict the final line, and delegates line checks on a new WM. This yields improved space-time trade-offs. As in TTCTR, the input trips need to be sorted by the same criteria, but in XCTR we use three complementary structures to represent each component of the sequence, as shown in Figure 5.4:

(i) An adapted CSA over the stop identifiers of all trips, concatenated into a string with additional terminator symbols \$ appended at the end of each trip. As in the CSA from TTCTR, we make these \$ symbols maintain the order of the trips and cyclical in Ψ. Because this time we do not encode line and stop identifiers together and CSA only encodes stops, there is no need for a complex vocabulary anymore.

(ii) `WML`: Aligned to the entries of (i) there is a WM for the line identifiers of each stop. Aligned to the \$ section we duplicate the starting lines of each trip. As a trivial optimization, we build a separate WM for every stop, allowing us to save space due to the fact that a single stop does not usually belong to many lines, thus the average height of these WM is no larger (and usually smaller) than the height of a single WM. In addition, since there are stops that belong to only one line, the corresponding `WML`s for those stops can be implicitly kept, and therefore they are not actually stored.

(iii) `WMJ`: A WM of jcodes aligned to last level of (ii). Note that this makes this structure dependant on (ii), which is coherent with the fact that journey codes themselves are

relative to the line identifier. In case (ii) implements the optimization described, the entries of this WM must also be rearranged to match the delimited stops.

Figure 5.4: An example of five trips represented on XCTR with the optimizations for `WML` and `WMJ`, and sections for each stop delimited by dotted lines.

Example 5.3: Just as in example from Section 5.2.2, in Figure 5.4 we build XCTR with five trips over the network shown in Figure 5.2. This time, unlike for TTCTR, our vocabulary V will only contain stop identifiers. These identifiers are used to build the CSA with our cyclic Ψ variation. Then, the sequence of lines is aligned to the entries of Ψ, and `WML` is built. Note that the only stops used in this example are 0, 3, 9, 10, 12 and 14. Because the stops 3, 12, and 14 belong to only one line each, we do not even need to keep a WM for their sections (which are delimited by D), as the value will always be the same. This corresponds respectively to the 2-nd, 5-th, and 6-th sections delimited in the figure. Finally, the sequence of Jcodes is aligned to the leaves of `WML`, and `WMJ` is built.

For example, $<9, 2, 1>$ from the fourth trip is represented as follows: the stop id 9 is mapped to $V'[\text{rank}(B, 9) - 1] = V'[2] = 2$, and it is encoded in T at position 13. It delimits the 10-th suffix in the sequence, as $A[10] = 13$, therefore its line identifier 2 also appears in position 10 in the sequence $Lines^{\Psi}$. Because in this example we have opted for the optimized version of `WML`, there is a single WM dedicated for the entries of stop 9, which fall in the section of $A[9..11]$, as delimited by D. Therefore, such `WML` represents the entries $Lines^{\Psi}[9..11] = <1, 2, 1>$ and consequently requires only one level that contains the bitmap 010. This means that in the last (conceptual) level of that WM would contain $<1, 1, 2>$ and consequently our entry for line 2 will appear on the third position. As the sequence $Jcodes^{WML}$ is aligned to this last level of `WML`, the journey code can be found at $Jcodes^{WML}[11] = 1$. □

In the later Section 5.3, we will detail how these structures are used in order to solve

the queries proposed in Section 5.1, and we will also compare the time complexities with those of TTCTR for every kind of query.

5.2.4 T-Matrices

While both TTCTR and XCTR are effective solutions for the trip pattern queries (type B in Section 5.1), they can be too unpractical to efficiently solve network load queries (type A). For this reason, we propose T-Matrices, a structure based on the SAT described in Section 3.1, to which we have designed a compression scheme that will reduce the total size of the structure while maintaining the $O(1)$ time bound for obtaining the sum of an arbitrary rectangle.

We build a SAT M_l^b for each line l, with a column for every stop of that line and a row for every journey, sorted by their starting times. For each cell $M_l^b[j, s]$ we store the number of users that have boarded on the stop s during the journey j from the line l. There is also an analogous matrix M_l^a that stores the number of alighting users in l. A small example of a single `T-Matrix` is displayed in Figure 5.5, where *Basic* refers to the raw (unaggregated) values, *Sum* are the accumulated values of a SAT, and *Blocks* is our compressed version.

Basic

2	1	2	1
1	1	2	3
3	2	1	1
2	1	2	0
1	0	0	3
3	1	0	1
0	2	1	0
2	1	1	1

Sum

2	3	5	6
3	5	9	13
6	10	15	20
8	13	20	25
9	14	21	29
12	18	25	34
12	20	28	37
14	23	32	42

Blocks

2	3	5	6
1	2	4	7
4	7	10	14
6	10	15	19
7	11	16	23
12	18	25	34
0	2	3	3
2	5	7	8

Figure 5.5: T-Matrices example.

In order to compress the size of T-Matrices, we observe that, because the amount of journeys is usually much larger than the amount of stops in the line, every T-Matrices is expected to be tall and narrow. Therefore, we can divide the matrix into blocks of r rows, so that only the first row of the block stores the absolute values of a SAT, while the values in the rest of the rows will be relative to the first one. Given a block matrix R built on the SAT S, any value from S can be then obtained in $O(1)$ time as $S[i, j] = R[\lfloor i/b \rfloor + i\%b, j] + R[i, j]$[2] when $i\%b \neq 0$, and $S[i, j] = R[i, j]$ otherwise.

[2]The symbol % refers to the modulo operation i.e., the remainder of the integer division.

5.3 Algorithms

In these section, we show how the structures from Section 5.2 allow us to solve the queries proposed in Section 5.1. Finally, we will provide a complexity analysis that we will later support with experiments in Section 5.4.

5.3.1 Solving network load queries

With T-Matrices, we can directly solve most of the proposed network load queries with summations over the matrices. For example, `board_X`$_L$, when we are not restricting to a time interval T, is equivalent of summing the corresponding column of the stop X for the matrix M^b_L. If we did not want to restrict to a line L either, since there is a M^b_l for every line l, we would simply have to add all the sums of those matrices whose lines contain the stop X. All this also holds true for `alight_X`$_L$, using the alighting matrix M^a_L instead of the boarding one.

When dealing with queries that do restrict to a time interval T, we must find the jcodes of the journeys that will fall within T, which can be done with the common structures discussed in Section 5.2.1, which store the initial time for each journey and also the average time of arrival to each stop of a line. An example of these structures being used to find a range of jcodes can be found at Algorithm 2, where the function `lower_bound` is an exponential search that returns the index of the first occurrence that is no lesser than the queried value, while `upper_bound` returns the index of the last no greater occurrence. We also use $lineStop_l^{-1}$(s) to obtain the position of a stop s within the sequence of stops from a line l.

```
1 Function GetJCodes(l,s,t_a,t_z):
      Data: line l, stop s, times t_a,t_z
      Result: jcodes for t_a and t_z
2     offset ← avgTime_l(lineStop_l^{-1}(s));
3     j_a ← lower_bound(initialTime_l, t_a-offset);
4     j_z ← upper_bound(initialTime_l, t_z-offset);
5     return < j_a, j_z >;
```

Algorithm 2: Obtaining the codes of the journeys from the line l that should arrive to the stop s within the time range given by t_a and t_z.

The range of jcodes obtained $[j_a..j_z]$ will correspond to a range of rows in the `T-Matrix` of the queried line, thus allowing us to solve `board_X`$_{LT}$ (or `alight_X`$_{LT}$) as a sum of the rectangle $M^b_L[j_a..j_z, s]$ (or $M^a_L[j_a..j_z, s]$), where s is the corresponding column for the stop X. Similarly, we can solve `use_L`$_T$ with a summation on the last column of L, restricted to the range of jcodes of T.

Finally, we can solve `load_X`$_{LT}$ by combining both the boarding and the alighting matrices, with the observation that at any stop X, the number of passengers in the vehicle has to equal the number of passengers that had boarded at or before X minus the number of passengers that had alighted. Therefore, for any line L, a range of stop indices $[s_a..s_z]$

that occur in L until X, and a range of jcodes $[j_a..j_z]$ obtained for T, it follows that

$$\texttt{load\_X}_{LT} = \sum_{s \in [s_a..s_z]} \sum_{j \in [j_a..j_z]} M_L^b[j,s] - M_L^a[j,s]$$

, which can be computed in $O(1)$ because we refer to a contiguous range in M_L^b and M_L^a.

5.3.2 Solving trip pattern queries

As with CTR, we also obtain a clear separation between the spatial representation of the trips (CSA) and the temporal representation (WM of $Jcodes^\Psi$) in TTCTR. The spatial component can be used to address queries such as "number of passengers that started their trip from a stop $X \in S$ and a line $l \in L$" ($\texttt{start\_X}_L$ from Section 5.1) with a binary search of the pattern $\$X_l$, as the pair $<X, l>$ will be mapped to a single symbol X_l in V. The temporal component can be used to filter down these results to a time window ($\texttt{start\_X}_{LT}$) with a $\text{count}_{j_a,j_z}(Jcodes^\Psi, i, j)$ operation over the WM, where j_a and j_z are *jcodes* obtained from Algorithm 2 and i and j delimit the range of the results obtained in Ψ. Because the \$ symbols were made cyclical in Ψ, it is also possible to answer `from_X_to_Y` queries by searching for a pattern $Y\$X_l$ instead.

With XCTR we can overcome the main weakness of TTCTR, i.e. that it requires several binary search operations over the CSA in the following cases:

- We are interested in the number of passengers that started their trip at a stop X and a time window $t_a..t_z$, but from **any line** ($\texttt{start\_X}_T$). As *jcodes* are relative to lines, we must make a separate query for each possible pair $<X, l_i> \forall l_i \in L$.
- We need to restrict the line of a final stop, in queries such as $\texttt{end\_X}_L$ or $\texttt{from\_X\_to\_Y}_L$ (and similar variations). Because the final stops belong to separate entries of the vocabulary that do not encode line identifiers, to restrict a stop $Y \in S$ to a line $l \in L$ we need to search for every possible expanded pattern $W_l, Y...$, for every stop W from the line l that could have been boarded before alighting at Y. While it looks tempting to address this issue by modifying the design of TTCTR so that final stops also encode line identifiers, this would in turn make queries that do not restrict the line of the final stop inefficient, and we would need to perform a new query for every combination of (Y, l_i) $\forall l_i \in L$, as in the previous case.

We address these limitations in XCTR, at the expense of an additional structure, which involves more (conceptual) complexity in our operations. We will now proceed to detail how a single trip t_i may be extracted from this compact representation in Algorithm 3, where $\texttt{WML}(\texttt{s}_a)$ is the WM corresponding to the stop $\texttt{s}_a$ in the optimized version of `WML`[3] and `TrackDown` returns the leaf index of a WM given a root index. In a practical implementation, it is not needed to access `WML` and `WMJ` for the line identifier and jcode of the last stop of the trip, as they will always match the previous ones.

Note that Algorithm 3 starts with the first stop of the trip, for which it recovers the corresponding line l_a and journey j_a. Then, it continues with the next stop of the trip in line 10, until the ending \$ (a 0) is reached.

[3]Recall that the optimized `WML` keeps a separate WM for every stop, instead of a single one for all the line identifiers. Without this optimization, the pseudocode would still be valid by assigning $z \leftarrow 0$ at line 6.

```
1  Function Extract_trip(i):
       Data: trip number i
       Result: Sequence of tuples <s, l, j> that compose the trip
2      trip ← [];
3      a ← Ψ[i];
4      s_a ← rank_1(D, a);
5      while s_a ≠ 0 do
6          z ← select_1(D, s_a);
7          l_a ← WML(s_a)[a − z];
8          j_a ← WMJ[TrackDown(WML(s_a), a-z) + z];
9          append < s_a, l_a, j_a > to trip;
10         a ← Ψ[i];
11         s_a ← rank_1(D, a) − 1;
12     end
13     return trip;
```

Algorithm 3: Extracting the trip i from XCTR, using its components Ψ, D, `WML` and `WMJ` from Figure 5.4.

A more complex example of these structures working together is the query "number of trips that started from a stop s_a and ended at a stop s_z", which can be further restricted to specific starting and ending lines and a time window (`from_X`$_{LT}$`_to_Y`$_{LT}$). The pseudocode for the full version of such query can be found in Algorithm 4. This algorithm relies heavily on the $\text{count}_{j_a,j_z}(Jcodes^{\Psi}, i, j)$ operation, as well as its variant count^{LR}, both defined in Section 3.4.

We will now proceed to explain the process in Algorithm 4 line by line:

- In **line 2**, we query our CSA for the pattern consisting of the destination stop s_z, followed by a \$ and finally the origin stop s_a. This results in a range of entries within the section of s_z that belong to our queried trips, because the trips were made circular in Ψ, so each \$ points to the beginning of its own trip. If we were not interested in restricting lines nor time, the function would end here, returning right-left.
- In **lines 3-5** we obtain the corresponding range in the section of \$ by accessing Ψ. Note that because of how the sorting of the \$ symbols was altered during the construction of the suffix array, these two ranges are equal in size, as the comment in line 5 points out.
- In **line 6** we query `WML` in the \$ section, within the range previously obtained, for the queried starting line l_a, obtaining the range of its occurrences $[a..z] \subseteq [left_0..right_0]$, since the \$ were sorted by their starting stop first, ending stop second and their starting line third. Note that if the XCTR were constructed without the optimization that separates `WML` in sections, this line would be exactly the same, except for the

```
1  Function FromXtoY_full(l_a,l_z,s_a,s_z,t_a,t_z,n):
       Data: lines l_a,l_z, stops s_a,s_z, times t_a,t_z and length of the sequence n
       Result: Number of occurences
2      [left, right] ← bsearch(Ψ, s_z$s_a);
3      left_0 ← Ψ[left];
4      right_0 ← Ψ[right];
5      // right-left = right_0-left_0
6      [a, z] ← count^{LR}_{l_a,l_a}(WML($), left_0, right_0);
7      // left_0 ≤ a ≤ z ≤ right_0
8      [j_a, j_z] ← GetJCodes(l_a,s_a,t_a,t_z);
9      [a, z] ← count^{LR}_{j_a,j_z}(WMJ, TrackDown(a), TrackDown(z));
10     a' ← TrackUp(WML($),a);
11     z' ← TrackUp(WML($),z);
12     // z-a = z'-a'
13     offset ← select_1(D, s_z);
14     [a, z] ← count^{LR}_{l_z,l_z}(WML(s_z), left − offset + a' − left_0, left − offset + z' − left_0);
15     [j_a, j_z] ← GetJCodes(l_z,s_z,t_a,t_z);
16     return count_{j_a,j_z}(WMJ, offset + a, offset + z);
```

Algorithm 4: Querying for `from_X`$_{LT}$`_to_Y`$_{LT}$ with all restrictions on XCTR.

query being on a WML that would encode the whole sequence of lines instead of just the WML($) for $.

- In **line 8** we obtain the range of jcodes for the journeys from the line l_a that would pass through the stop s_a within the time window delimited by t_a and t_z, using the function `GetJCodes` from Algorithm 2.
- In **line 9** we operate over a range of `WMJ` that encodes a non-decreasing sequence of jcodes, given that within the same origin stop, final stop, and starting line, the $ symbols were sorted by the starting journey code of their trips. This allows us to keep using count^{LR}, as we are still operating within the $ section. We also need to use `TrackDown` on a and z since `WMJ` is aligned to the last level of `WML`.
- In **lines 10-12** we use the `TrackUp` operation, which is the inverse of `TrackDown`: it returns the position in the first level given a position in the last level of a WM. In this case, as the a and z we obtained in the previous step are also positions in the last level of the `WML`, we use `TrackUp` to translate that range of indexes to the root of `WML` and therefore to the entries of the CSA as well. Note that the resuling range is inside the $ section for trips with the same origin stop, final stop, and starting line that includes all the trips for jcodes that span from j_a to j_z. The properties of our adapted CSA also ensure that this range maintains the same size after translation and it can also be directly translated to a range in [left..right].
- In **line 13** we obtain the starting position of the section for the stop s_z in the CSA with a select$_1$ operation over the bitvector D. This is necessary for operating on `WML`

and WMJ to restrict the line and journeys for the final stop.

- In **line 14** the range [a'..z'] $\subseteq$ [left$_0$..right$_0$] is trivially translated to the section for s_z within [left..right], where we query WML to obtain the subrange [a..b] $\subseteq$ [left..right] for the line l_z. Remember that WML(s_z) represents only the lines for s_z, therefore both the translated positions and the resulting subrange positions are relative and must be adjusted by offset. This would have not been necessary if the optimization was not implemented, and absolute positions would have been used (offset$\leftarrow$ 0).

- In **line 15** we obtain the jcode range analogously to line 8, but this time for l_z and s_z.

- In **line 16** we return the number of entries of WMJ between j_a and j_z within our final subrange. Unlike count^{LR}, count does not report any range boundaries, but simply returns the number of occurrences.

It is trivial to reuse Algorithm 4 to answer queries with less constrains. For example, if we were only interested in restricting the starting line, we could return z-a after line 6. If we only wanted to restrict the ending line and time, we could do it by skipping the lines 3-12, and using directly count$^{LR}_{l_z,l_z}$(WML(s_z),left-offset,right-offset) in line 14.

The complexity would increase if we restricted by a time window but not by lines, as we would need to iterate through all possible lines for s_a and s_z to obtain the jcodes for each line and perform these operations on WML and WMJ. Fortunately, the number of lines a single stop can belong to tends to be rather small in practice, thus with a careful implementation that avoids repeating computation, the performance of such query scales well, as it will be shown later in Section 5.4. Finally, to obtain only the trips that started on a given stop we would simply need to set pattern to $\$s_a$ in line 2, or alternatively to $s_z\$$ for the final stop, and skipping the operations on the sections of s_z or \$, respectively.

Additionally, XCTR can also be used to efficiently obtain other interesting information about trips, such as the top k most boarded stops, with the possibility of differentiating stops that are only used to switch lines in XCTR. However, to the best of our knowledge, there exists no efficient way of using this representation to obtain other kinds of information efficiently (e.g. the number of passengers in a journey between two stops).

5.3.3 Analyzing our representations

To illustrate the application of each representation, as well as highlight the motivation for the development of XCTR, in this section we discuss in detail the worst case time complexities of each query described in Section 5.1, being l the number of lines we restrict to, $|L|$ the total number of lines represented, $|S|$ the number of stops in the queried line

and $|J^l|$ is the maximum number of jcodes per line.

	Query	T-Matrices	TTCTR	XCTR
Type A	`board_X`$_{LT}$	$O(l)$	$O(l \cdot \log(\lvert J^l \rvert))$	$O(\log(n\lvert L \rvert \cdot \lvert J^l \rvert))^{\otimes}$
	`alight_X`$_{LT}$	$O(l)$	Hard$^{\ddagger\diamond}$	Hard$^{\ddagger\diamond}$
	`use_L`$_{T}$	O(1)	$O(\lvert S \rvert \log(\lvert J^l \rvert))$	$O(\lvert S \rvert \log(n\lvert L \rvert \cdot \lvert J^l \rvert))^{\otimes}$
	`board_T`	$O(\lvert L \rvert))$	Hard$^{\diamond}$	Hard$^{\diamond}$
	`alight_T`	$O(\lvert L \rvert)$	Hard$^{\ddagger\diamond}$	Hard$^{\ddagger\diamond}$
	`load_X`$_{LT}$	$O(l)$	Hard$^{\ddagger\diamond}$	Hard$^{\ddagger\diamond}$
Type B	`start_X`$_{LT}$	-	$O(\log(n\lvert J^l \rvert))^{\otimes}$	$O(\log(n\lvert L \rvert \cdot \lvert J^l \rvert))^{\otimes}$
	`end_X`$_{LT}$	-	$O(\lvert S \rvert \log(n\lvert J^l \rvert))^{\otimes}$	$O(\log(n\lvert L \rvert \cdot \lvert J^l \rvert))^{\otimes}$
	`switch_X`$_{LT}$	-	$O(\log(n\lvert J^l \rvert))^{\otimes}$	$O(\log(n\lvert L \rvert \cdot \lvert J^l \rvert))^{\otimes}$
	`from_X`$_{LT}$`_to_Y`$_{LT}$	-	$O(\lvert S \rvert \log(n\lvert J^l \rvert))^{\otimes}$	$O(\log(n\lvert L \rvert \cdot \lvert J^l \rvert))^{\otimes}$
	`start_L`$_{T}$	-	$O(\lvert S \rvert \log(n\lvert J^l \rvert))^{\otimes}$	$O(\log(\lvert L \rvert \cdot \lvert J^l \rvert))$
	`end_L`$_{T}$	-	Hard$^{\diamond}$	$O(\lvert S \rvert \log(n\lvert L \rvert \cdot \lvert J^l \rvert))^{\otimes}$
	`start_T`	-	Hard$^{\diamond}$	$O(\lvert L \rvert \log(\lvert L \rvert \cdot \lvert J^l \rvert))$
	`end_T`	-	Hard$^{\diamond}$	Hard$^{\diamond}$

$^{\otimes}$ The complexity will increase when the line is not restricted. See discussion.
‡ May include false positives.
$^{\diamond}$ Not practical to solve with the indexing capabilities of this representation.

Table 5.1: Worst case time complexities for the representations described in Section 5.2, assuming the queries have all the restrictions.

Before explaining the details of how each of these operations would be implemented in our representations, it is worth noting that the complexity of the XCTR and TTCTR queries marked by $\otimes$ will depend on the constraints that are applied, as explained in the previous Section 5.3.2. Generally, the time complexity of XCTR with all the restrictions is bounded by one bsearch operation on the CSA to restrict by a stop id, one count^{LR} on `WML` to restrict by a line id and, finally, a count on `WMJ` to restrict by jcodes, with a total complexity of $O(\log(n|L| \cdot |J^l|))$, being n the size of the represented sequences.[4] However, when only the time is restricted, every line that the queried stop belongs to must be considered, thus increasing the worst-case complexity to $O(\log(n) + |L| \log(|L| \cdot |J^l|))$. Similarly, the complexity of TTCTR is normally a search in the CSA and a count on the WM (if time is restricted), which would amount to $O(\log(n|J^l|))$, although when only the time (and not the line) is restricted, we must perform a new query for every possible line, and the complexity increases to $O(|L| \log(n|J^l|))$, which is generally higher than $O(\log(n) + |L| \log(|L| \cdot |J^l|))$, the one from XCTR.

- `board_X`$_{LT}$. This can be solved easily in the T-Matrices, as there are aggregated matrices for boarding stops, even though it is necessary to access a separate matrix for every line queried, hence $O(l)$. For TTCTR, the number of occurrences for the

[4] While theoretically locating a pattern of length m in a Suffix Array takes $O(m \log n)$ time, in our case m is bounded to 2 or 3 (in case of `from_X_to_Y`). Furthermore, with a backward search implementation, only $m - 1$ binary searches are needed.

stop X is counted by delimiting its range in D (with a $O(1)$ select_1 operation) and filtering down the WM if needed. For XCTR, the filtering is done through `WML` and `WMJ`, and after that we must subtract the occurrences of final stops, obtained by one `end_X`$_{LT}$ with the same restrictions.

- `alight_X`$_{LT}$. Both in TTCTR and XCTR, the only alighting stops that are explicitly represented are the final stops. To obtain an approximated count of the rest of them, we would need to extract and examine all the occurrences for `board_W`$_{LT}$ for every stop W that could have been boarded before X, as the worst-case time complexity of solving with their index operations is prohibitive. On the other hand, it can be solved much faster in T-Matrices by accessing the alighting matrices.
- `use_L`$_T$. This operation is straightforward to solve in T-Matrices, as it only needs to access one matrix. For TTCTR we must filter through the WM for every stop from L. While in XCTR the total number of occurrences of line L can be calculated in one $O(\log |L|)$ operation (with additional filtering through `WMJ` if needed), we need to subtract the occurrences of `start_L`$_T$ and `end_L`$_T$, the latter having a higher complexity of $O(|S| \log(n|L| \cdot |J^l|))$.
- `board_T and alight_T`. Although T-Matrices, TTCTR, and XCTR could solve it by applying `use_L`$_T$ $|L|$ times, this would imply a prohibitive cost for both TTCTR and XCTR, where extracting all the trips would be preferred in practice.
- `load_X`$_{LT}$. It is possible to solve this kind of query using both alighting and boarding matrices in T-Matrices. Knowing how many travelers got on and off the vehicle previously to one particular point makes trivial to determine how many of them are in the vehicle between two stops. This method can be easily adapted to measure the average number through an interval of time.

The following Type B operations are not supported by T-Matrices:

- `start_X`$_{LT}$. We need to the range for the pattern $\$X$ in the CSA and to filter down the WMs, if needed.
- `end_X`$_{LT}$. In XCTR it is solved similarly to `start_X`$_{LT}$, by delimiting the range for the pattern $X\$$ in the CSA and to filter down WMs, if needed. It can be much more complex for TTCTR, where the only way to restrict a line is to make a new query for every stop that could have been boarded before X in that line, which is a problem detailed at the begining of Section 4.
- `switch_X`$_{LT}$. It is calculated as `board_X`$_{LT}$ - `start_X`$_{LT}$.
- `from_X`$_{LT}$`_to_Y`$_{LT}$. It was explained in detail in Algorithm 4. In TTCTR the $|S|$ factor only takes effect if the line (or time) for Y must be restricted, as for `end_X`$_{LT}$.
- `start_L`$_T$. It is trivially solved in XCTR, by filtering down `WML` and `WMJ`, if needed, over the \$ section of the CSA. On the other hand, for TTCTR the CSA needs to be queried with the $\$X$ pattern for every stop X from line l.
- `end_L`$_T$. It can be solved in XCTR, by performing a `end_X`$_{LT}$ for every stop X from the line l. If we used the same strategy for TTCTR, we would end up with a complexity of $O(|S|^2 \log(n|J^l|))$, which we consider prohibitive.
- `start_T`. It is solved in the XCTR by applying `start_X`$_{LT}$ $|L|$ times.

- `end_T`. None of our representations can solve this query efficiently, although it could be approximated by `start_T`.

5.4 Experiments

In this section we discuss the practical performance of our structures. To evaluate them, we have run randomly generated queries against T-Matrices, TTCTR, and XCTR built over a dataset of synthetic user trips generated from a real transportation network (Section 5.4.1), with several configurations to study the trade-off between compression (Section 5.4.2) and query efficiency (Section 5.4.3), and testing different configurations for each individual structure.

5.4.1 Experimental dataset

Using real GTFS[5] descriptions of bus routes and schedules, we generated a synthetic dataset of user trips that aims to realistically imitate real user behaviour during a month. In this work, we have combined the GTFS obtained for the networks of urban[6] and interurban[7] buses for the city of Madrid. The network model was extracted and user trips were generated with the following general steps:

1. We parsed stop, route, and trip identifiers. This produced two lines per route in almost all cases, one for each direction. With the gathered data, we were able to build the common structures *lineStop* and *stopLine* discussed in Section 5.2.1.
2. We connected stops that are on a short walking distance (100 meters) from each other, or appear sequentially on the same line.
3. We parsed the schedules for bus trips.
4. We generated a month of journeys from the schedules, differentiating days of week. From this step, we computed $avgTime$ and $initialTime$ common structures.
5. User trips were generated. A trip starts from a random stop, day, and journey and simulates boarding that journey and traversing it. After each traversed stop, the user may end the trip with a probability that starts at zero and increases by 1% for every stop visited. Additionally, there is a fixed probability of attempting to switch lines at the current stop, if there is a journey available at that stop within the allowed waiting time (30 minutes) and from a different line.[8] Switching lines is also attempted when the end of the current line is reached.
6. We persisted these generated trips as sequences of stages
   $<$ $<line, journey, boarding\_stop>$, $<line, journey, alighting\_stop>$ $>$,
   where a boarding and alighting stop naturally share the same line and journey within the same stage. This results in the same number of stages as lines have been used. With the parameters used, about 56% of our trips have one stage, 33% have two, 9% have three, and 2% have four.

[5] `https://developers.google.com/transit/gtfs/`
[6] Provided by EMT `http://www.emtmadrid.es`
[7] Provided by CRTM `http://www.crtm.es`
[8] The reverse of the current line is also disallowed.

With this approach we have generated a dataset of ten million trips, over a real network consisting of 11021 stops, 1048 lines for a simulated month,[9] with an average of 1622 journeys per line and a maximum of 9980 per line. We consider that this synthetic dataset is of enough accuracy and size to obtain significant results when studying the compression capabilities and performance of our representations.

5.4.2 Space requirements

We have measured the in-memory sizes of all the individual components of our representation built over the experimental dataset. We also present the sizes of our common structures, followed by T-Matrices. After that, we compare the compression achieved with TTCTR and XCTR.

For the CSA structures from TTCTR and XCTR, we use an adapted *iCSA* from [FBN+12], and tuned it with the t_Ψ (sampling interval) factors of 32, 128, and 512. For the WM present in TTCTR, as well as in the two WM from XCTR, we analyze the space required by four different configurations: when we tune the WMs to use the `RRR` bitvector and tuning it to use a plain bitvector with a rank structure, which we call `RG32`.

The space occupied by the common structures is reflected in Table 5.2. These were represented using plain fixed bit length integers with the exception of $initialTime$, where we used the compression approach discussed in Section 5.2.1 with a sampling interval of 512. This has allowed us to represent all required network information in a negligible space of less than 1 MiB.

Structure	$lineStop$	$stopLine$	$avgTime$	$initialTime$	Total
Size (KiB)	119	141	64	440	764

Table 5.2: Sizes of the common structures.

To measure the amount of space occupied by the two different variants of T-Matrices that were presented in Section 5.2.4, we have summed the total space required by each line matrix. These results, which can be found in Table 5.3, helped us to prove that `Blocks` is an effective strategy to reduce the size of the accumulated values.

Variant	`Sum`	`Blocks`
Size (MiB)	55.49	28.68

Table 5.3: Sizes of the two different variants from T-Matrices.

In Table 5.4 we analyze the space occupied by the two structures that compose TTCTR: the CSA that encodes stops and their lines, and the WM that encodes journey codes.

We have observed that when we use a very sparse configuration of t_Ψ, the CSA becomes a small representation, as it captures the repetitiveness of trips when constrained to our network of bus lines.

[9] A period of 31 days, starting on a Monday.

(a)

t_Ψ	bps	Size (MiB)
32	7.288	30.91
128	6.069	25.74
512	5.761	24.43

(b)

Bitvector	bps	Size (MiB)
RG32	14.438	44.02
RRR32	13.478	41.09
RRR64	12.79	38.99
RRR128	12.446	37.94

Table 5.4: Space requirements for the Compressed Suffix Array (a) and the Wavelet Matrix (b) from TTCTR.

We were not able to achieve such good level of compression for the WM, where compressing the bitvectors with `RRR` results in a WM that is only slightly smaller than the baseline with the `RG` plain bitvector. This sequence is indeed hard to compress, given than it was rearranged to be aligned to the entries of our CSA, which nullifies any local redundancy that other kinds of arrangements may obtain.

We have also measured the space occupied by XCTR in Table 5.5, where three structures must be taken in consideration: the CSA that only encodes stop identifiers, and two WM: one for line identifiers (`WML`) and the WM for journey codes (`WMJ`).

(a)

t_Ψ	bps	Size (MiB)
32	6.956	29.50
128	5.727	24.29
512	5.417	22.97

(b)

Bitvector	bps	Size (MiB)
RG32	14.438	61.23
RRR32	13.419	56.91
RRR64	12.732	53.99
RRR128	12.388	52.53

(c)

Bitvector	bps	Size (MiB)
RG32	4.778	20.26
RRR32	2.338	9.91
RRR64	2.18	9.24
RRR128	2.103	8.92

Table 5.5: Space requirements for the CSA (a), the `WMJ` (b), and the `WML` (c) from XCTR.

When compared to the structures of TTCTR, we can observe that `WMJ` of XCTR, despite achieving a marginally better compression than the WM from TTCTR, requires considerably more space, as the sequence of jcodes from XCTR includes final stops, while in TTCTR they are skipped.

Contrary to the case of `WMJ` or the WM from TTCTR, we are able to achieve significant compression with `WML`. When using the baseline bitvector `RG32`, we are already obtaining a very compact representation, due to the optimization discussed in Section 5.2.3, where we

keep a separate WM aligned to each stop from the CSA, resulting in much shorter WMs. The further compression achieved with the `RRR` bitvectors in XCTR is possible because the stop entries in the CSA are sorted by either the final stop the user alights to or the next stop to board, making the aligned sequence of lines very predictable. Additionally, our CSA maintains the order of trips from the original text in the $ section, leading to the appearance of clusters in the first `WML`.

In Table 5.6 we also evaluate the compression of the overall TTCTR and XCTR representations with respect to a plain representation of the user trips that we use as input, which are the triplets $<s, l, j>$ seen in Section 5.2.2, where the stop identifiers, line identifiers, and journey codes had to be represented with 14-, 11-, and 14-bit integers, respectively. We only show the compression ratios for the four configurations that will be tested in the following Section 5.4.3, combining two t_Ψ and two WM configurations.[10] Note that a more realistic baseline would be a relational database representation, where additional fixed-width integers would be needed to maintain foreign key relations, thus requiring much more space than our chosen baseline.

		plain		TTCTR		XCTR	
t_Ψ	WM	MiB	%	MiB	%	MiB	%
32	RG32	165.39	100	81.73	49.42	111.73	67.56
32	RRR128			75.66	45.75	91.70	55.44
512	RG32			75.25	45.50	105.21	63.61
512	RRR128			69.18	41.83	85.17	51.50

Table 5.6: Sizes (in MiB) and compression ratio of TTCTR and XCTR shown as a percentage of the size of a plain representation of the user trips with fixed-width integers.

We can see that when compared to the same baseline, the compression ratios of XCTR are worse than those of TTCTR, for any configuration tested. This result is not surprising considering that XCTR separates the line identifiers into a different WM, that has proven to be less compressible than a CSA for repetitive sequences, although we have also observed that while the CSA is slightly smaller in XCTR, the compression ratios are worse than the ones in TTCTR when we compare to a smaller baseline that only represents the sequence of stops, with no information about lines. In the next section we will show the advantages of this separation of line identifiers in terms query performance.

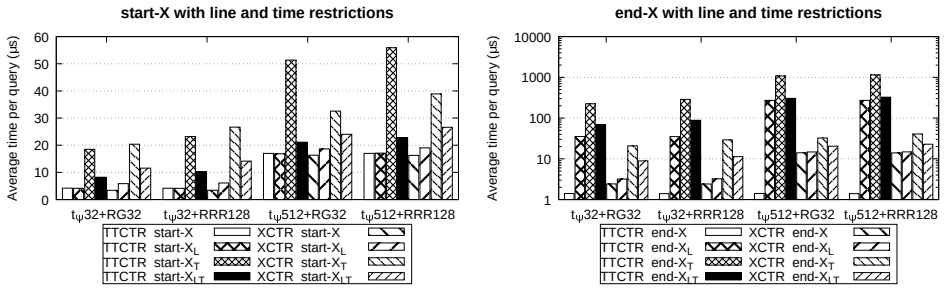
5.4.3 Query performance

We have implemented the most adequate queries for TTCTR and XCTR from those compared in Section 5.3.3, and measured their average execution time from 100,000 randomly generated queries on a Intel Xeon E5-2620v4@2.1 GHz machine, running Debian 6.3.0 and compiling our code with GCC 6.3.0 using the `-O3` optimization flag. In this section we will only discuss four of the possible configurations tested for both TTCTR and

[10] In case of XCTR, this refers to bitvectors from both `WML` and `WMJ`.

XCTR. We tuned the CSA to use $t_\Psi \in 32, 512$ and configured the WMs to use either uncompressed bitvectors (`RG32`) or the most compressed bitvector setting (`RRR128`). These four configurations should be illustrative enough to provide an understanding of the space-time trade-offs of our approaches. Finally, we will also compare the performance of the query `board_X`$_{LT}$ with the one achieved by the T-Matrices.

In Figure 5.6b, we can see the main advantage of XCTR over TTCTR, where restricting a line or a time interval for an `end_X` query is very expensive for TTCTR due to its separate vocabulary for final stops. Recall it requires to query the CSA for every possible stop that could have been boarded before X to restrict a line. In Figure 5.6a, it is also possible to see how the performance of XCTR degrades less with a highly compressed CSA for the query `start_X`$_T$, where even queries over a compressed `WML` are faster than queries over the CSA.



(a) **(b)**

Figure 5.6: Comparison of `start_X`$_{LT}$ (a) and `end_X`$_{LT}$ (b) queries, with all variants. Note the logarithmic scale for the y axis in (b).

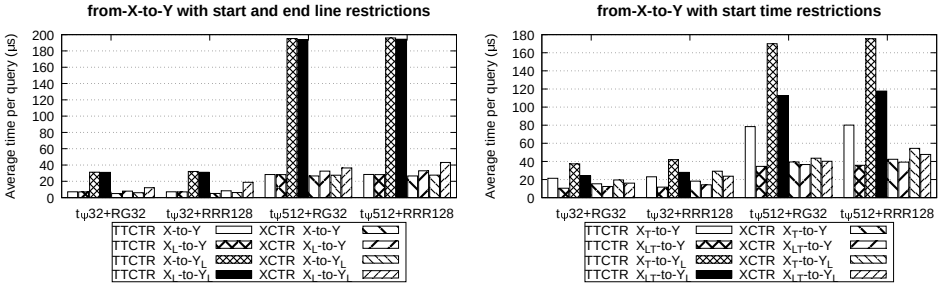
Recall that restricting both the line and time is always cheaper than only restricting the time, as for the latter more operations need to be performed to filter out every line. This explains why T queries are always slower than LT queries. This is true for both representations, with any configuration and query.

We can see more examples of this difference in performance between our two representations with the `from_X_to_Y` queries in Figure 5.7, whenever the end lines or times are restricted. Additionally, we can observe yet again how using the most sparse sampling of Ψ affects much more the performance of TTCTR than that of XCTR, with the performance of `from_X`$_T$`_to_Y` consistent with the `start_X`$_T$ shown in Figure 5.6a.

The performance of both representations can sometimes improve when more selective restrictions are added, where the execution is cut short when no matching trips are found, before evaluating further restrictions. For this reason, the average times for `from_X`$_T$`_to_Y`$_L$ and `from_X`$_{LT}$`_to_Y`$_L$ are faster than those of `from_X_to_Y`$_L$ and `from_X`$_L$`_to_Y`$_L$.

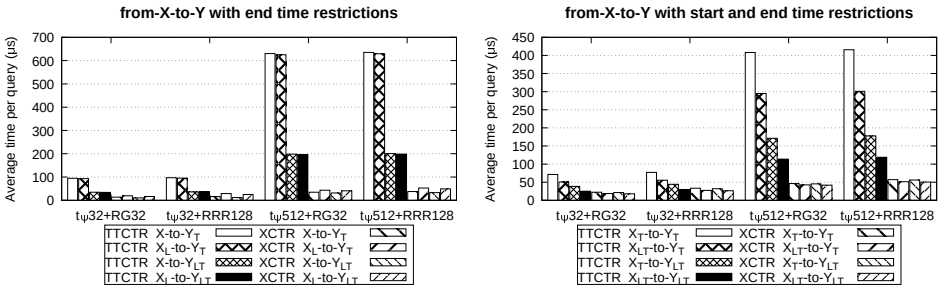
When restricting to the end time, XCTR consistently outperforms TTCTR, as shown in Figure 5.8. This was expected considering the large number of times that the CSA from TTCTR needs to be queried, yielding results similar to those from Figure 5.6b where the end time is also restricted.

The high selectivity of time restrictions explain why TTCTR becomes more competitive with the most restrictive queries of Figure 5.8b. However, its query time still increases



(a) **(b)**

Figure 5.7: Comparison of `from_X`$_{LT}$`_to_Y`$_{LT}$ queries, variating line (a) and starting time (b) restrictions.



(a) **(b)**

Figure 5.8: Comparison of `from_X`$_{LT}$`_to_Y`$_{LT}$ queries, variating line (a) and starting time (b) restrictions with a fixed ending time restriction.

several times when the CSA is highly compressed.

The only query for which TTCTR is clearly preferred over XCTR is `board_X`, with any restriction, as it can be seen in Figure 5.9a. Both for `board_X` and `board_X`$_L$, TTCTR takes on average less than one microsecond per query, as the only operations needed are two constant time select$_1$ over the bitvector D from the CSA, while XCTR needs to subtract the occurrences of $X\$$ (as those are alighting stops, not boarding), for which Ψ must be accessed. This advantage is carried on the queries with time restrictions as well.

When comparing the best performing configuration of TTCTR with the different variants of T-Matrices discussed in Section 5.2.4, we can observe wildly different results in Figure 5.9b depending on the variant of the `board_X` query used. The biggest difference is observed when comparing queries that filter by time, which use the WM in TTCTR, while any variant of T-Matrices solves it in a small number of $O(1)$ operations. Another evident difference occurs within each T-Matrices variant, where the queries that are restricted to a single line are significantly faster than those that consider every line, as the latter must query a different matrix for every line where the stop occurs in. It was also expected to see the compressed variant `Blocks` to perform slower than the uncompressed `Sum`, as the

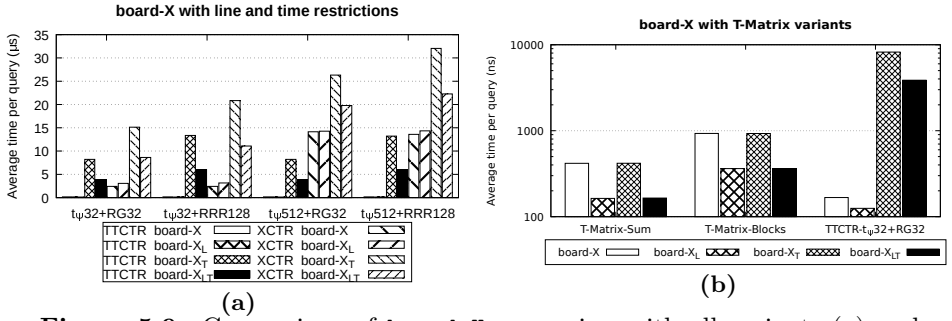


Figure 5.9: Comparison of `board_X`$_{LT}$ queries, with all variants (a) and also with all variants of T-Matrices (b). Note the logarithmic scale in (b), as well as the measurements in nanoseconds.

two former store relative values that must be resolved, increasing the number of memory accesses, although still by a constant factor. This also hints for a reason why the queries `board_X`and `board_X`$_L$ seem to be slightly faster in TTCTR than any of the T-Matrices, as a higher number of accesses over larger memory regions in T-Matrices makes cache misses more frequent. Nevertheless, the difference is very small, within the same microsecond.

Obviously, T-Matrices may be used to solve other queries more efficiently than TTCTR or XCTR. We did not find it interesting to report the run times for those queries as they are mostly equal to those shown in Figure 5.9b, due to being resolved with the same CPU operations. Refer to the complexities in Table 5.1 to obtain an accurate estimate of the time that it would take T-Matrices to solve each of the supported queries.

Part II

Towards a GIS-based application to analyze trips data

Chapter 6

Previous concepts on GIS

Any application aiming at providing a simple and usable user-interface in the scope of the analysis of data from a transportation scenario should, without any kind of doubt, consider a map-based user interface. Therefore, in this chapter we shall introduce the basic concepts of Geographic Information System (GIS) in order to provide a context for this part of this thesis.

While there exist several complementary definitions for GIS in the literature, from the point of view of the current work we will regard a GIS as an application designed to assist a decision making process related to spatial or geographical information [HA03, LGMR15].

In this chapter, we initially discuss the most significative features of spatial data. After that, we focus on conceptual models for geographical data on Section 6.2, and Section 6.3 is devoted to review logical models. The last two sections present the main GIS standards we used both to make spatial data accessible for a real GIS application and also to visualize that data on a GIS interface.

6.1 Spatial information features

One of the main challenges in GIS is the treatment of spatial information, which presents some distinctive features that make it more complicated to store, manage and visualize than the kinds of data managed by traditional information systems (e.g. an accounting database). We consider that the most significative features of spatial information are:

- **Two alternative conceptual interpretations.** Geographical information may be regarded on a conceptual level from three different perspectives: as geographical objects, as geographical fields, or as topological networks. This division on an abstract level will also propagate to the more concrete levels.
- **Multiple possible logical models.** The complexity of geographical information makes it possible to have several possible and valid representations for the same geographical phenomenon. Each one of them with their own advantages and disadvantages depending on the context.

- **Data types and specific operations.** Due to the three possible conceptual interpretations and the multiple possible logical models, there are numerous possible datatype sets. Moreover, there also exist many different operations that may be defined over these datatypes, and there is no standard for a minimum amount of operations that may be used to define every other operation.
- **Complex and varied methods of analysis.** Depending on the field of application where the GIS is used on, there may exist a great variety of techniques to analyze geographical information that are expected to be included in the GIS.
- **Large datasets.** Geographical information can be quite voluminous, both in the complexity of every element and in the number of elements that must be stored. As a result, the structures used to store this information have to be specific.
- **Slow transactions.** The complexity of geographical information can make updating times significantly slow, which in turn can make long locking times on the stored elements. A GIS must consider updating mechanisms, since its information is naturally dynamic.
- **Spatial indexing**. Due to the large size of the datasets, combined with the spatial features (i.e. number of dimensions, space boundaries, and overlapping), the indexing structures that are used to provide an efficient access method to the stored information are particular to a specific use case.
- **Implicit hierarchy.** Unlike traditional information, geographical information is always visualized over a bounded space (the map). Because of this, the visualization scale acquires a main role in the presentation of the information and also determines an implicit hierarchy, as it is not possible to display all the information at the same time.
- **Special visualization techniques.** A usable interface to visualize geographical information must include commonly expected features, such as interactive controls, a layer abstraction and an acceptable performance to be considered responsive.

6.2 Conceptual models

A first step to define a conceptual model for geographical information is to be able to measure and understand the geographical space. To achieve that, we need a mathematical definition of space and to define a Coordinate Reference System (CRS).

6.2.1 Coordinate Reference Systems

In order to represent a spatial element, we need to define a CRS, because such representation will depend on our mathematical definition of space. One common coordinate system is the Cartesian one, based on an Euclidean space, a plane over which bidimensional objects may be represented. However, this system is insufficient for broad areas of the Earth's surface, where the curvature of the Earth must be taken into account. To address these cases, the geometry of the planet is often approximated with a spheroid or an ellipsoid, over which spherical coordinates (latitude and longitude) are used instead of Cartesian

ones. Currently, the most popular coordinate system is based on the **WGS84** ellipsoid,[1] used for GPS navigation.

Once the definition of space has been established, we need to choose a coordinate system and define how it will reference the space (i.e. origin of coordinates, orientation and scale). Depending on the application, we can directly use the spherical coordinates defined by the ellipsoid or project these coordinates into an Euclidean space. The most common projection to work with relatively small areas of the Earth's surface is the Universal Transverse Mercator (UTM), which divides the chosen ellipsoid into 60 zones, each one of them spanning 6 degrees of longitude. Within each zone, it is possible to project the curved surface into a plane with minimal precision loss,[Kar11] thus allowing to reference coordinates in a Cartesian system, with a scale of meters, where the first component is the Easting from the central meridian (plus an offset of 500,000 meters to avoid dealing with negative numbers) and the second component is the Northing from the intersection between the central meridian and the Equator.

Example 6.1: Using a CRS defined by the WGS84 ellipsoid, the building of the Faculty of Informatics of the Universidade da Coruña is located at the coordinates (43.332709,-8.410517). Another possible CRS can be defined by projecting the WGS84 ellipsoid using UTM, which would place the longitude of the building (-8.410517) into the zone 29, with the coordinates (547788, 4797931). This projection may also be used with a different ellipsoid, yielding a different CRS with some other coordinates. □

6.2.2 Abstractions for geographical information

Once the CRS is defined, we must define the abstractions that will allow us to represent geographical information, which can be considered from three alternative perspectives: as geographical objects, as geographical fields, or as topological networks.

Geographical objects are subsets of space that are used to represent the position or extension of spatial entities. The surface of a road or the position of a store are examples of geographical objects. Figure 6.1 (left) shows an example of geographical objects on a map.

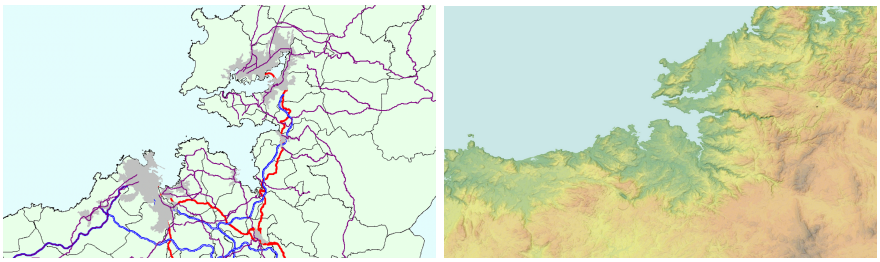


Figure 6.1: Two of the possible abstractions for geographical information: geographical objects (left) and geographical fields (right).

[1] `https://epsg.io/4326`

Note that there can be several valid representations for a geographical object, regardless of its extension in the real world. For instance, a city may be represented as a single position (point) or as a geometric surface, depending on the ultimate requirements of visualization or exploitation.

A **geographical field** is characterized as a function that associates a value for every point of space. The surface temperature or the slope of the terrain may be considered as geographical fields. An example of a geographical field can be found in Figure 6.1 (right), where an elevation map is shown.

Both abstractions described above (geographical objects and geographical fields) can be alternatively used to represent information of different nature. Commonly, geographical objects are used to analyze man-made structures, such as roads, borders, or land registry, while geographical fields are more appropriate for natural information collected by sensors, such as meteorological, geological, or satellite information. However, this does not mean that, for example, a building could not be represented as a geographical field that assigns a value of 1 for a coordinate contained within the building and 0 for every other coordinate, although such conceptual definition may be ill-fitted for many applications.

There are defined standards for modeling geographical objects, such as ISO 19107,[2] based on primitives objects (points, curves, and surfaces) that may be combined to form more complex types. In addition, ISO 19123[3] is a standard used for modeling geographical fields, which may be discrete (existing only a value for some discrete points in space) or continuous, having a value for any point in space.

As a special case, it is also possible to work with a topological network, where instead of using a coordinate system, we can model our space as a graph. An example of such application may be a road network used for GPS routing, where the movement is restricted by roads instead of by a true Euclidean space. While there are also standards for these abstractions, ad-hoc models based on vertices and edges are more commonly used instead.

6.3 Logical models

In the previous Section 6.2, we have defined conceptual models, that work on an abstract level. However, the abstract models do not regard the limitations of a practical implementation. As such, they may assume that both the memory and arithmetic precision are infinite, and consequently a geographical object is an infinite set of points while a geographical field is a defined function for any point in space. These limitations must be addressed in order to implement a GIS, and for that purpose we must work with a logical model.

In this section, we will present two well-known models used to represent geographical information, although we will not cover how the most used operations are implemented using these models or how the precision problems have to be addressed. Refer to [Xia15] for a more in-depth discussion on those topics.

[2] `https://www.iso.org/standard/26012.html`
[3] `https://www.iso.org/standard/40121.html`

6.3.1 Vector model

This model uses numerical data types (such as integers or floating point representations) to define a system of coordinates over which the geographical information is represented using geometric constructions (points, segments, polygons, etc...). For each primitive data type defined in the geographical object conceptual model, there is a vector representation.

The most extended vector model is the Simple Feature Specification, defined by the Open Geospatial Consortium (OGS),[4] in which data types are defined to represent simple spatial objects defined in the ISO 19107. The simpler data types are composed to represent more complex ones, as shown in Figure 6.2, where a *curve* is approximated as a *LineString*, which is defined as a sequence of *points*, while a *surface* can be a *Polygon* defined with *LinearRings*, which are special cases of *LineStrings* where their last *point* is equal to the first one, thus enclosing a space.

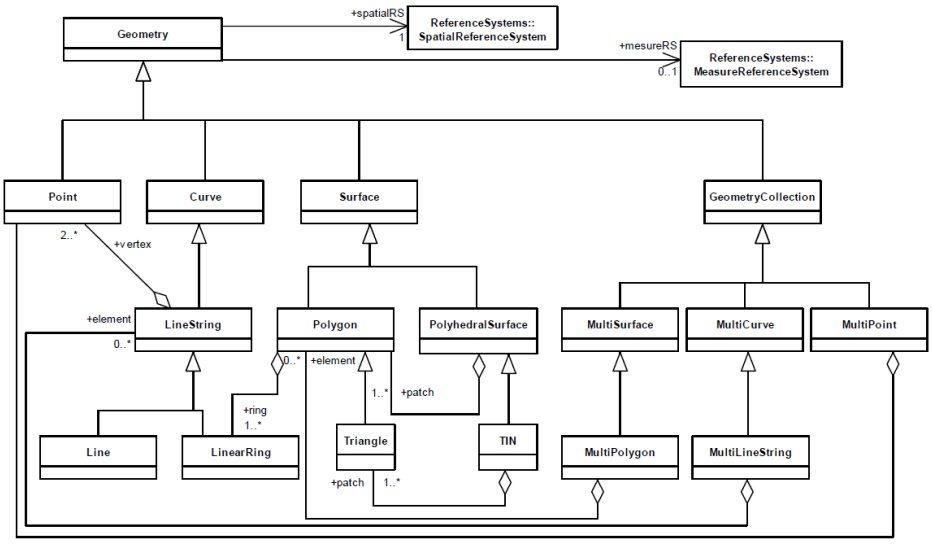


Figure 6.2: The data types of the Simple Feature Specification by the Open Geospatial Consortium

This composition can be also seen in Figure 6.3, where six *points* are used to define a *LineString*, while three *LinearRings* are used to define a *polygon*, one for the external borders of the *polygon* while the other two define the internal borders (the *holes*).

[4] `https://www.opengeospatial.org/standards/sfa`

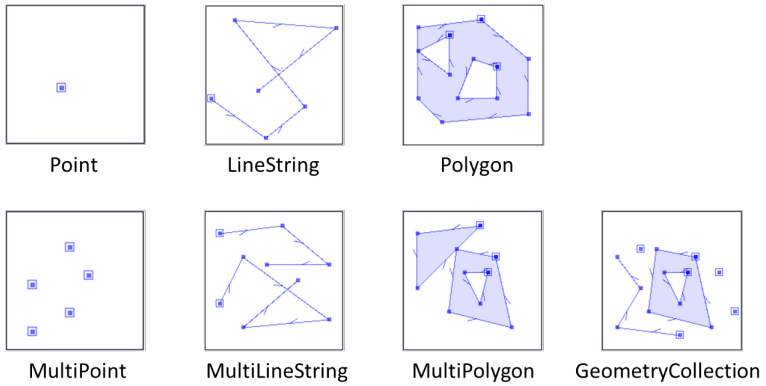


Figure 6.3: Examples of Simple Feature Specification data types.

6.3.2 Raster model

The raster model represents the spatial information using a matrix, where every cell stores a value for a portion/cell of the space. The meaning of each value will depend on the particular case. To cite some examples, the value of a cell may refer to temperature, pressure, humidity, wind speed, or the color of the surface. In order to translate a cell position to coordinates in a CRS and vice versa, we need six values: the position of the origin of coordinates and how much the height and width of each pixel contribute to the x and y coordinates in the CRS. Since the raster may be oriented in any angle, a single dimension in the matrix (such as the width) may contribute to both the x and y components in the CRS.

Rasters are frequently stored within an image format with some additional metadata for the CRS transformation. An extension of the *TIFF* format called *GeoTIFF*[5] is one of the most common formats for these purposes, as it allows to define a CRS and the linear transformation parameters for coordinates. Additionally, it is possible to store several images in a single TIFF file, which is often used to store several levels of detail for the raster, allowing to speed up operations that do not require the highest resolution.

6.3.3 Comparison of vector and raster models

In the previous section we have described two possible logical models to represent geographical information. Both the vector and raster models can be used to represent either geographical objects or geographical fields. As previously seen, each model has a different approach to the practical limitations when implementing a conceptual model. In this section we will show how each of the logical models can represent either geographical objects or geographical fields and also make a brief comparison of both models.

[5] `https://www.opengeospatial.org/standards/geotiff`

In the vector model, geographical objects are represented as a *discretization*, which may lead to some precission loss in some of the shapes, as seen in Figure 6.4 (top). In the other hand, the raster model can represent objects by a process called *rasterization*, where each cell value is assigned to an object identifier, as seen in Figure 6.4 (bottom).

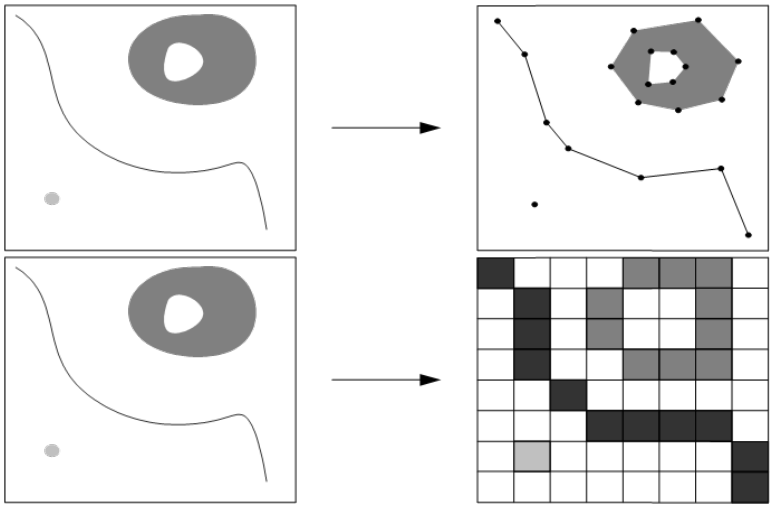


Figure 6.4: Representing geographical objects in the vector (top) and raster (bottom) models.

Geographical fields are also transformed when they are represented in a logical model. As such, in the vector model, these fields are represented with a *polygonization* of the function, as shown in Figure 6.5 (top), while in the raster model we discretize the function, representing a unique value only for the cells of the matrix, as done in Figure 6.5 (bottom).

As both models are interchangeable, the best choice will depend on the specific GIS application intended. The decision can be influenced by the representation size (vector model is usually more compact), the processing efficiency (raster algorithms are often more efficient and straightforward than the vector ones), expressive capabilities, visualization (the precision of the vector model vs the efficiency of the raster one) or source of the information.

6.4 Standards

Over the years, many GIS standards have been developed and adopted by the community, making it easier to develop and integrate applications using multiple data sources. In this section, we are only going to cover the standards used by the GIS prototype *Trippy* (see Section 7).

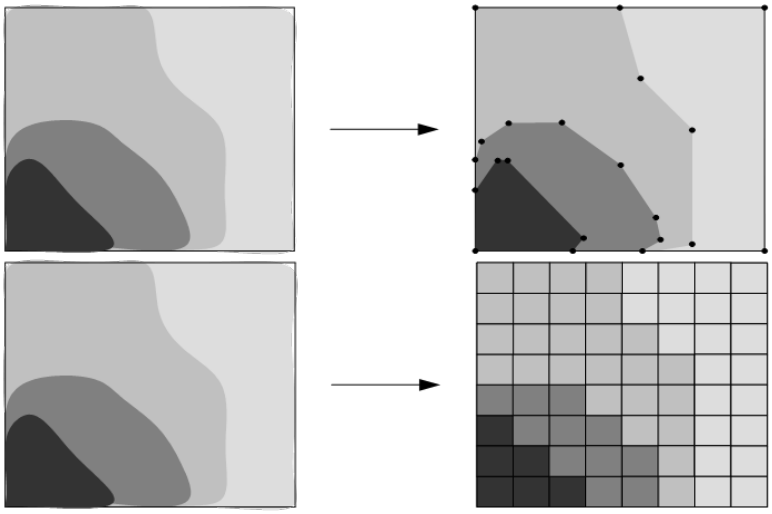


Figure 6.5: Representing geographical fields in the vector (top) and raster (bottom) models.

6.4.1 TMS

The Open Source Geospatial Foundation defines a standard to generate cartographic images (tiles) called Tile Map Service (TMS),[6] which aims to simplify and modernize some aspects of an older standard called Web Map Service (WMS).[7]

A TMS service returns map parts, called *tiles*, at several levels of zoom with *HTTP GET* requests, typically in PNG format, although other formats can be supported. With a special metadata request that returns the definition of the CRS and the parameters to translate pixel coordinates to CRS coordinates (discussed in Section 6.3.2) at all the supported zoom levels, it is possible to develop a web-based map application, loading tiles interactively as a user spans and zooms over the map.

It is also possible to integrate TMS from several sources, as tiles can have transparent background. This could allow, for example, to paint tiles containing only bus lines over tiles showing the streets of a city. Due to its ease of implementation, both for server side and for client libraries, this standard has been widely adopted and made available by operators such as OpenStreetMap, Google, Thunderforest, and many others.[8]

[6] `http://wiki.osgeo.org/wiki/Tile_Map_Service_Specification`
[7] `https://www.opengeospatial.org/standards/wms`
[8] `https://wiki.openstreetmap.org/wiki/Tile_servers`

6.4.2 GeoJSON

While there is a well-known web standard for accessing geographical features defined by the Open Geospatial Consortium called Web Feature Service (WFS),[9] which allows to obtain, among many other kinds of information, the vector representation of the geometry for these geographical features, this standard is often found too complex for the single purpose of transmitting vector geometries.

For this reason, custom `REST APIs` that encode vector geometries in GeoJSON[10] (RFC 7946) have become the most popular way of transmitting this type of data. This format makes it possible to encode *Points*, *LineStrings*, *Polygons*, and their multipart variants in a compact string, while also allowing to attach any arbitrary information (*properties*) to these features in JSON format.

6.4.3 GTFS

Developed as a standard to represent the schedules of most public transportation systems, the General Transit Feed Specification (GTFS)[11] was developed by Google and it provides a standard model to specify stops, routes, schedules, and other useful information for trip planning that may be associated with a transportation system, as shown in the diagram from Figure 6.6.

This has some resemblance to the model we have previously proposed in Figure 5.1 from Section 5.1, where a `stop` in GTFS corresponds to a `stop_place` in our model, `stoptime` to `stop_time`, a `trip` is similar to a `journey`, while a `route` is a `line`. The largest difference, however, is that we consider that a line (route) is formed by a fixed sequence of stops, while in GTFS no such restriction exists, and every trip can theoretically have a different sequence of stops.

There are GPS coordinates stored for a stop, and it is also possible to define a *LineString* shape for a trip, which will be referenced by a `shape_id` from a shapes definition file. The stop times for a trip are usually defined within the calendar dates, allowing this way to specify a schedule based on the day of the week or have a special consideration for selected days. Outside of the calendar definition, it is still possible to specify an expected frequency that a trip must respect.

6.5 GIS interfaces

In the previous sections we have discussed how to represent geographical information in conceptual and logical models, as well as what GIS standards are followed in this work. In this section, we will briefly speak about how this geographical information can be presented to the user and visualized in a GIS interface, as well as introduce a web mapping library called *Leaflet*.[12]

[9] `http://www.opengeospatial.org/standards/wfs`
[10] `https://geojson.org/`
[11] `https://developers.google.com/transit/gtfs/`
[12] `https://leafletjs.com`

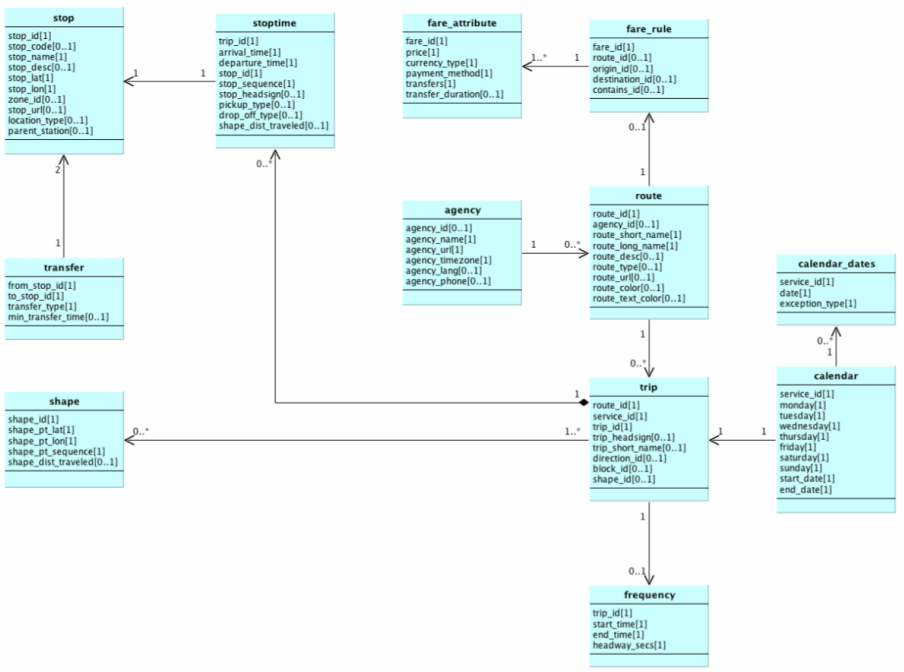


Figure 6.6: UML class diagram of the entities specified by GTFS.

6.5.1 Data visualization

The visualization of geographical information is often complex due to the special features of spatial information. In the first place, these features require a different abstraction level than the ones used by traditional databases. In the second place, this information usually needs to be transformed with a projection in order to be able to visualize it on a flat surface. It is also necessary to define a *visualization metaphor* for the user interface: the stored geographical information is presented to the user in several layers.

A layer can be seen as a set of geographical elements that are intended to be visualized with a common style. These layers are later sorted in a stack, to determine which layers are superior or inferior to others. That way, the information of the superior layers is presented over the information of the inferior ones.

This system of layers allows to display a digital map in a similar fashion to that of a paper map. However, a GIS user interface is fundamentally different from a conventional map, as the information and visualization is no longer static. Hence, the interface needs to support interactive operations that can alter this visualization. Among these operations, we would like to mention the following ones:

- Layer management, with the possibility of adding or removing layers, as well as reordering them in the stack.

- Scrolling over the map, defining a new point of origin.
- Changing the scale of the visualization, increasing or reducing the zoom level.
- Invocation of processing operations over the map. These may include, among others, interactive measurements or queries about the visualized elements.

Another important element in this visualization is the context information, which must be included within the user interface in order to better understand the displayed information. This context information frequently includes a graphic legend, which allows to interpret and categorize every geographical element. Additionally, the interface must have means to inform the user of the current visualization scale and the center coordinates. Other information may be necessary, such as labels that identify geographical objects or present relevant information.

6.5.2 Leaflet

Among the several available alternatives to develop a web-based map, in this work we have used *Leaflet*, which is an open source `Javascript` library that can be integrated with different data sources and providers to easily develop interactive maps with standard web technologies (`HTML`, `CSS`, and `Javascript`).

It supports both the TMS and the *GeoJSON* standards described in the previous Section 6.4, among others. This combination is often used to display a base map from a tile provider with vector information layers over it obtained in *GeoJSON* format.

Although this library is comparatively lacking in features when compared to other well-known alternatives such as Google Maps[13] or *OpenLayers*,[14] it is frequently preferred due to its usability and smaller size. Due to its broad adoption, open source plugins exist for all the features offered by the other comparable solutions, including support for other data sources (such as WFS), advanced features (such as routing), and also small interface tweaks (such as the integration of a context menu).

[13] `https://developers.google.com/maps/documentation/`
[14] `https://openlayers.org/`

Chapter 7

A GIS interface for public transportation networks

While in the previous part of this work we have presented compact representations that are capable of efficiently handling large collections of trips over both streets and public transportation networks, in this chapter we present a front-end interface that permits them to be actually usable by city management and network administrators.

We have developed a proof-of-concept map-based interface using GIS and web technologies called **Trippy**, that allows to exploit some of the capabilities of XCTR to visualize and query transport demand information over a real transportation network. This constitutes a tool designed to aid the decision-making process for a public transportation company, and can be easily adapted for different networks and use cases.

In Section 7.1, we begin by contextualizing the use case in detail. After that, we provide an overview of the architecture of *Trippy*, both from a functional perspective and a technical one. After that, we present our API for querying the underlying structure, and finally we show the user interface of our prototype.

7.1 Motivation and overview

In public transport, such as buses or trains, we find that most transport companies have been focused mainly on providing helpful information to their passengers regarding the existing offer, hence providing not only information related to their transport network (e.g. maps with the lines, their schedule, etc.) but also, in many cases, real-time information with the actual position of a vehicle, remaining time to destination, remaining time until next vehicle arrives, etc. However, although there is also an increasing interest on gathering actual information regarding what users demand from the public transport and how they use their services, to the best of our knowledge, there are no final end-user tools that have tackled both the problems of: *(i)* effectively handling the huge amount of data that arises when we track all the user's trips (e.g. within a large city) and *(ii)* efficiently exploiting the underlying information.

Therefore, we are focusing on a solution prototype that allows both to query and visualize historical records of aggregated patterns of movement from the travelers of a transportation network. While there exist popular solutions such as Google Maps to accurately calculate optimal multi-modal routes and even monitor the real time condition of the transportation networks, these systems only manage information about the *offer* of the network (such as schedules) and sometimes the real-time *demand* (current traffic conditions). In contrast, our approach aims at satisfying query needs about the historical demand over a large collection of past trajectories collected from users.

When dealing with vast amounts of historic traveler data, a valid first approach to tackle its analysis is to extend a traditional Database Management System (DBMS) that could both assimilate and handle greater volumes of spatial data. A popular tool that fits in this description would be *postGIS*,[1] a spatial extension for the widely adopted relational DBMS *PostgreSQL*. However, these solutions that include spatial indices are not the most adequate ones for our information needs, that are many times unrelated to a geographical component, but to the network elements and their relationships ("how many travelers started their trip at a stop X using line L to end it at a stop Y within the time frame T?"), while the expected large sizes of our data collections would force us to use solutions based on secondary memory, considerably affecting the performance of the system.

Given that we are focusing on a tool specifically for public transportation networks, our interface is built over the XCTR (designed in Section 5.2.3), which has allowed us to achieve fast response times in our tested use cases. In any case, the architecture designed for our tool makes it flexible enough to allow data to be represented in a more traditional way, such as with a relational DBMS.

7.2 Architecture

In this section we discuss the overall architecture of Trippy, our interface based on XCTR. We start by describing it from a functional perspective, following the data flow from the data sources until the presentation layer in the user interface. We complement this description with a technical architecture overview, speaking about how the individual components are integrated together and facilitate an understanding of the decisions behind our technological choices.

7.2.1 Functional architecture

We have designed Trippy to be as flexible as possible, ensuring that it can operate with any network and trip descriptions that can be adapted to the model from Figure 5.1, which is elementary enough to apply to most of the current transportation systems. For an overview of this data flow, refer to the diagram in Figure 7.1.

Although our system is oblivious to the specific data sources, we have successfully experimented adapting information collected from GTFS sources, as well as OpenStreetMap, to obtain a description of a public transportation network. Each different source or format would require a custom conversion to adapt it to our model, which is one of the reasons that we had for standing against overcomplicating our model. As for the passenger trips,

[1] `https://postgis.net/`

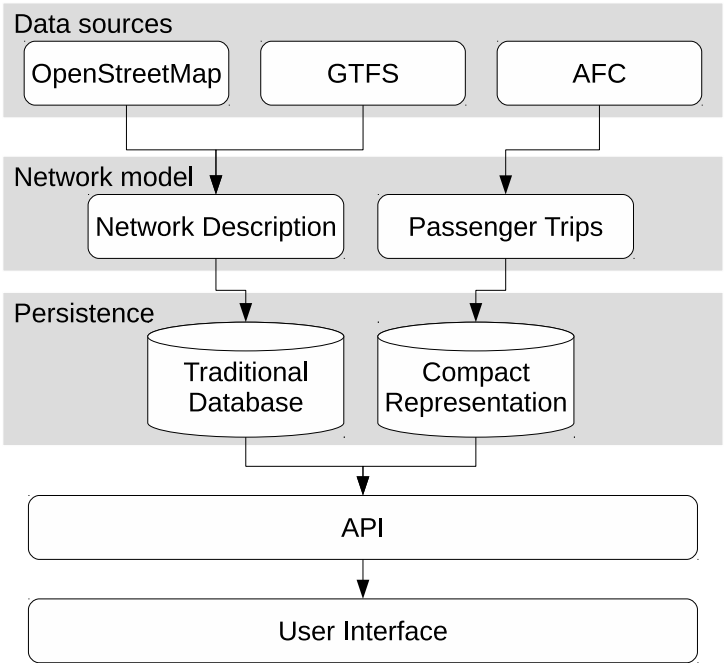


Figure 7.1: Functional architecture of Trippy.

they must be described in terms of stages (boarding and alighting pairs) that are related to stops and journeys. The amount of preprocessing needed will depend on the source of the data, as most Automated Fare Collection (AFC) systems do not directly record the alightings and the specific journey within the day will have to be estimated based on timestamps.

We have also opted for different persistence options for the network description and the passenger trips, as we are interested in using our XCTR for the latter, while the former does not pose any technological challenges that would benefit from the use of a compact representation.

Finally, these two data repositories are used to feed an API over which requests can be made, which is going to be exploited by the user interface. The specific supported requests are discussed in Section 7.3.

7.2.2 Technical architecture

We have designed and implemented the infrastructure needed to solve our query needs, which are focused on the usage of network elements, as previously explained in Section 7.1. To visualize and query on these network elements, we need to work with a representation of the network itself. Specifically, we represent the model for trips over a public transport

network shown in Figure 5.1, extended with spatial attributes (gps coordinates) for the stops and lines. Implementing this model for networks does not pose any challenge, as this information is rather static (it is not expected to grow significantly) and only requires a small amount of space in comparison to passenger trips. Therefore, a Relational Database Management System (RDBMS) can be used to represent the elements of the network. On the other hand, we use our compact representation XCTR for the trips, that we are going to rely on for most of our queries. The reduced size of this autoindexed representation allows us to keep it in primary memory, making it outperform any traditional indexing alternative that is based on secondary memory.

The overall technical architecture of our proposal is presented in Figure 7.2. In the bottom part, we can find the backend that includes two sources of information: the former one includes the network representation and relies on a small read-only database implemented in SQLite;[2] the latter source includes a XCTR, for our compact representation of the passenger trips.

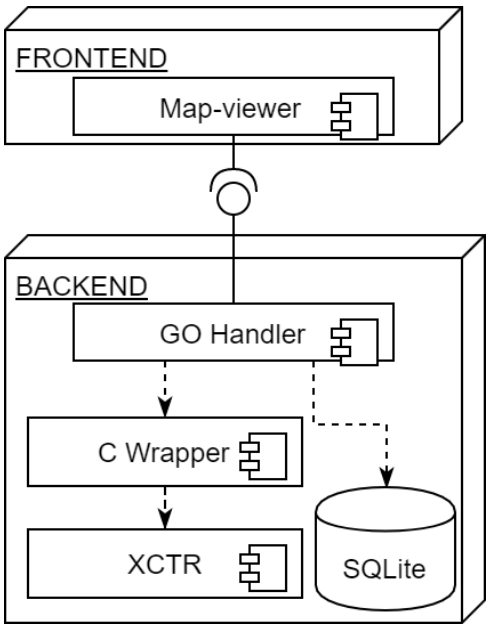


Figure 7.2: Technical architecture of Trippy.

The backend is implemented in the Go language,[3] and provides a uniform API to query the information sources. This is accessed by our GIS frontend (top of Figure 7.2), which is

[2] `https://sqlite.org/index.html`

[3] Since XCTR is implemented in C++, a C wrapper allows us to call the query functions supported by XCTR from the Go handler.

in charge of displaying the transport network on a web map and allows the final user to make queries by interacting with the elements of the map and its controls.

Further details about the elements included in both the backend and the frontend of Trippy are discussed below.

7.2.2.1 Representing the transport network (SQLite)

Since we aim at visualizing the elements of the network over a map, we need information regarding the stops and lines of the transport network, following closely the model from Figure 5.1. For each physical **stop**, along with the GPS coordinates of its location, we store a unique numeric identifier. Regarding the **lines** of the transport network, we basically keep the sequence of stops each line traverses. Note that, unlike the abstractions used by most public transportation networks, we consider lines to be one-way (i.e. they have a unique direction), and consequently the reversed/returning path is considered a different line.

As indicated above, we have chosen a SQLite database to represent the network because it is conveniently portable and efficient for such small datasets. While no spatial database technology was required to deal with the GPS coordinates of the stops, we have opted to integrate the SpatiaLite extension[4] to support storing the shapes of our lines.

7.2.2.2 GO handler

This component is on top of the backend of *Trippy* and receives the requests from the front-end. It has two main parts. On one hand, it implements a *REST API* that is used to obtain the collection of stops and lines in GeoJSON format. On the other hand, it also provides a querying interface for XCTR. While the query functions themselves are implemented in XCTR using *C++*, we have opted for *Go* as our main backend language because it is specifically oriented for web services and can be easily integrated with *C* code. In our case, we have implemented a thin *C wrapper* that interacts with the XCTR libraries to load XCTR structures (when the application is launched it loads the XCTR-based representation of the trips into memory) and handle calls to the query functions.

7.2.2.3 Map viewer

The Map viewer in Trippy makes up the frontend and provides the user web interface. It is in charge of visualizing the elements of the network, and handles user interaction. Apart from typical functions to move around a map (i.e. zoom in/out, span, etc.), the user can interact with the elements within the map (e.g. a stop or a set of stops within a region) and access functions related to them. Those functions are supported by our backend.

The Map viewer was built with a web map using the *Leaflet* library. With this library, it is simple to make an interactive map consisting of several interchangeable image (tiles) and vector-based layers, and also to efficiently represent thousands of points or other vector-type elements. For additional controls, communication with the backend, state synchronization

[4] `https://www.gaia-gis.it/fossil/libspatialite/index`

among our implemented controls and the map, we have used Vue.js,[5] which is a popular web framework based on components.

7.3 Our API to query transport-related data

In this section, we discuss the main query features that were integrated into *Trippy* and were defined within our API that is described below.

As previously seen in Section 5.3.3, XCTR supports a wide set of query functions, which have allowed us to solve a broad range of transport-related queries in *Trippy*. Those include, yet are not limited to:

- How many passengers boarded or alighted on a particular stop during a specific time interval (e.g. this evening, yesterday)?
- How frequently is a stop X used to switch lines during rush hours?
- How many passengers started/ended their trips at a specific stop X?
- How many passengers started their trips at stop X and ended them at stop Y during a given time interval?
- How many travelers had to switch lines to get to their destination during a given time interval?
- How many times a line was used as a start or end of a trip? Alternatively, how many times was the line used to switch between two other lines, even though it was neither the origin nor the final destination?

Along with these basic queries supported originally by XCTR,[BFG+18] we have extended its functionality to also support a range of more complex queries that were solved as a combination of the previous ones:

- How many passengers traveled from area A to area B during a time interval?
- At what time is a specific stop more crowded?
- At what time does a line get more crowded?

To support these queries, we have developed API Endpoints that we are going to discuss in the rest of this section. Note that a single request will frequently be responsible for more than one kind of query in the XCTR backend, as it is often more practical and efficient to perform multiple types of queries over the same element in XCTR for one single API request.

7.3.1 Stops endpoint

This endpoint is implemented as an *HTTP GET* request with the format `/stop/X`, where `X` corresponds to the queried stop identifier. This request returns the following information of the stop `X`:

- Number of passengers that had boarded in that stop.

[5] `https://vuejs.org`

- Number of times the stop was used to start a trip.
- Number of times the stop was used to end a trip.

Additionally, this request accepts parameters to restrict the *line identifier*, *lower time limit* and *upper time limit*, which will filter the numerical results mentioned above.

When this endpoint is queried with no parameters (`/stop/`), it will return the list of all the stops in the network, which is needed to display them on a map and initialize the user interface components. Consequently, the information for each stop returned by this query will consist of the following fields:

- Identifier.
- Name of the stop.
- Identifiers of the lines this stop belongs to.
- Geographical coordinates of the location of the stop.

7.3.2 Lines endpoint

We created a line endpoint that handles *HTTP GET* requests in the form of `/line/X`, being `X` the identifier of a line. Similarly to the previous endpoint, the information returned is also the equivalent to the one returned for the stops, but concerning a whole line instead of a single stop. Therefore, we can return:

- Number of passengers that had boarded (and alighted).
- Number of times the line was used to start a trip.
- Number of times the line was used to end a trip.

This endpoint accepts filtering by a time interval with the parameters of *upper and lower time limits*.

There is also a listing request `/line/`, to obtain a list of existing lines, which includes the following fields:

- Identifier.
- Name of the line.
- Identifiers of the stops this line contains, in order.
- If exists, the path of the line, encoded as a sequence of coordinates.[6]

[6]This information is optional, without it a line on the map can be displayed by simply connecting points with a straight line (which may not follow any road or track) or using an existing image tile (TMS) layer with lines.

7.3.3 Trips endpoint

This endpoint is dedicated to return one single field: the number of trips that have started at any of the stops from `X`* and finally end at any stop from `Y`*, where both `X`* and `Y`* can be either a single stop or a collection of them. Due to technical reasons,[7] this requests are implemented using the *POST* method over the `/trip` endpoint.

Both `X`* and `Y`* can be restricted by a *line* parameter (one line for starting stops and another for ending) and also *lower and upper times* (starting or ending times of the trip contained within the limits). These requests can be noticeably slow when the number of queried stops is very high (in the order of thousands).

7.3.4 Histograms endpoints

These are two endpoints that handle *HTTP GET* requests about time series of boarding events either for a stop (`/hstop/X`) or a line (`/hline/X`). The returned data is a sequence of pairs $<t_i, n_i>$, where each t_i is a timestamp and n_i is the number of boardings on t_i.

Those requests may be bound to a time window (by specifying *lower and upper time limits*, as in the other endpoints) and also accept an additional parameter of *sampling* that, if present, will lead to grouping the number of boardings by a given number of seconds s into bins. The query then returns a sequence of pairs $<t_i, n_i>$, where each t_i delimits the lower time limit of a bin that corresponds to the period $[t_i..t_i + s)$, and consequently, $t_{i+1} = t_i + s$. The second component of each pair, n_i is the total number of boardings within that delimited time period. A reasonably large sampling parameter may not only save bandwidth as less information needs to be transmitted, but also speed up the request.[8]

7.4 User Interface

Based on the API developed from Section 7.3, we have developed a prototype of an accessible and intuitive user interface, based on web application technologies. In the rest of this section, we present a kind of brief user manual that discusses the main elements of such user interface.

The interface of Trippy consists of a map as shown in Figure 7.3, where have set up a study case with the same bus networks from Madrid used in Section 5.4.1. Every bus stop is represented by a stop and we are using an underlying Thunderforest[9] TMS layer to display the bus lines that connect them. Additionally, the interface presents stop selectors ①, line selectors ② and also date ③ and time ④ filters that allow a user to query the XCTR for information.

As the interface shown is still in a prototype phase, it does not yet incorporate all the planned features. Concretely, we are not yet displaying the geometries of the lines nor making them interactive, and there is yet no way to make requests over the line or

[7] The requests may contain thousands of stops, surpassing the length limits of *GET* requests in many web servers, proxies and even modern browsers.

[8] In XCTR, a large sampling value may translate into more than one journey code for a line, thus making it possible to return a result before reaching the last level of the `WMJ` in the count operation.

[9] `https://www.thunderforest.com`

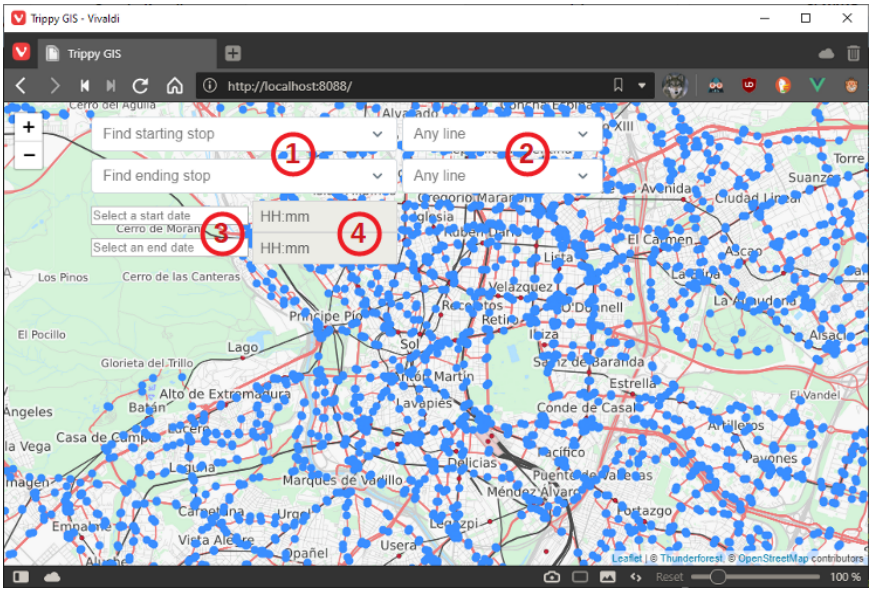


Figure 7.3: Main view of the user-interface of *Trippy*.

histogram endpoints from the previous Section 7.3 using the graphic interface. However, we consider it complete enough to be representative of the utility that such an interface can have when powered by XCTR.

By clicking on a stop, a popup is shows up with the stop name, the lines the stop belongs to, and also the number of users that used that stop for boarding, starting or ending a trip, or as a switch stop,[10] as shown in Figure 7.4 (left). These usage numbers will only be considered within the time window specified by the filter, if any. It is also possible to open a context menu for the stop with a right click, as done in Figure 7.4 (right). This makes up one of the ways to specify a starting or ending stop for an X to Y query (from the trip endpoint).

A future feature is planned to query the line endpoint in a similar way, in order to display usage statistics for a whole line instead of for single stops. Since several line paths may overlap, the interface must provide a way to disambiguate the selection. The navigation among stops and lines will also be helped by providing links in the corresponding popups.

In the upper part of the main screen we include stop and line selectors, which are implemented as custom dropdown components with a search field, as depicted in Figure 7.5. On the left box we are looking for stops containing *Bilbao* in their names, and on the right one we are checking lines from *Leganés*, from the partial query *legan*.

As in Madrid it is frequent to have several bus stops with the exact same name for

[10]The number of boardings must be equal to the number of starts added to the number of switches.

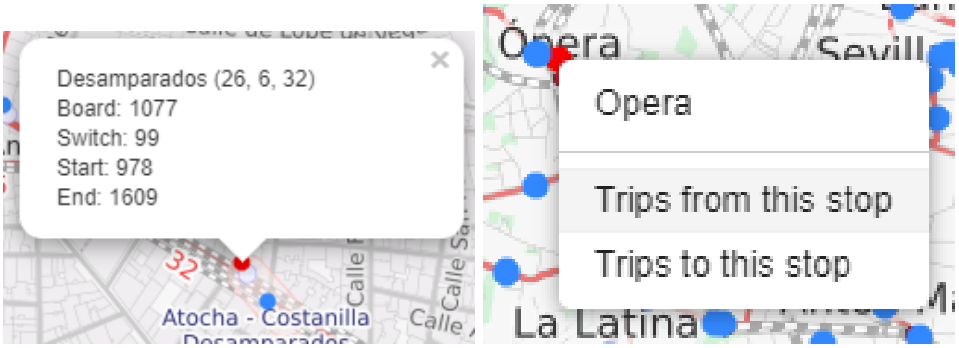


Figure 7.4: Stop popup showing the information and usage of a single stop (left), as well as its context menu (right).

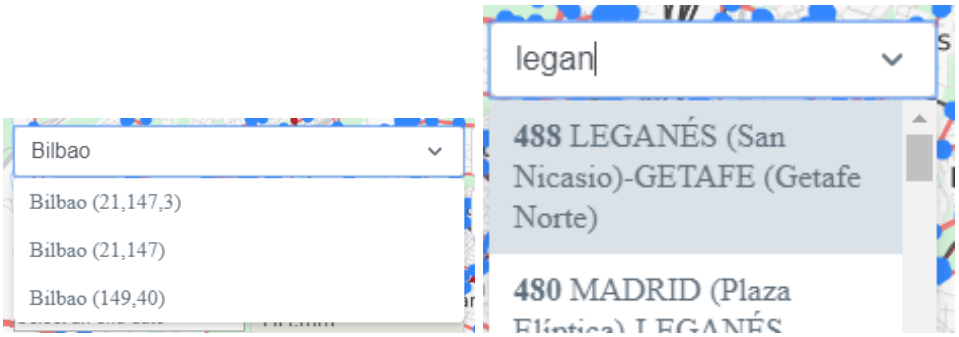


Figure 7.5: The stop (left) and line (right) selectors.

different physical stop locations, either because it is used by different lines or by the reversed direction of the same lines (as such, they tend to be located on the opposite direction of the same street), we display the lines that each stop belongs to, in order to help a user to distinguish between these cases. Note that in the displayed version we do not distinguish the direction of the line in the interface, although we do keep track of the directions internally, and consider the reverse direction to be a separate line. These two selectors are also correlated: selecting a stop in the starting (or ending) field will restrict the starting (or ending) line selector to only those lines that the selected stop belongs to, and vice versa.

The date and time filters affect all the displayed results, and can be edited with their corresponding components as shown in Figure 7.6. We also ensure that the selected starting and ending dates and times do not overlap: in our example only the dates starting from `2019-03-11` are available, since that is the selected starting date. If a popup is displayed while these filters are changed, the contents of the popup are dynamically updated.

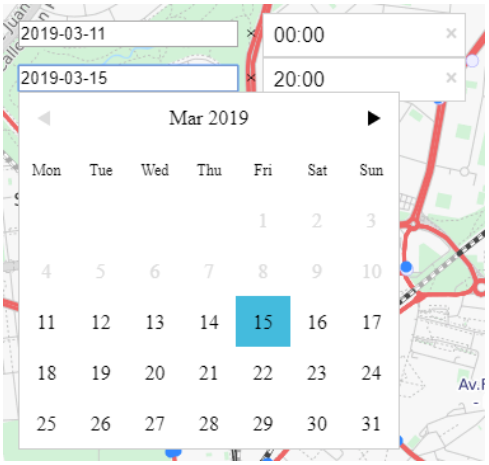


Figure 7.6: The date and time filters.

The trip endpoint (which returns the number of trips performed starting with a stop X and ending with a stop Y) may be queried by either using the starting and ending stop selectors or the context menu of the stops accessible with a right click, which also fills the selected stop into the corresponding stop selector. A straight arrow connecting the starting and the ending stops is displayed, as can be seen in Figure 7.7, and a popup with the total number of trips is also shown over that arrow.

Additionally, we allow both of these stops be be filtered by line, and the results filtered by a time window. These controls may be interacted with while the popup is open, and the number of results will be updated.

We can also perform the generalized version of this query, and obtain the number of movements between a starting and a final areas, as done in Figure 7.8.

In order to select the stops within a rectangular area, the user must press the SHIFT-KEY on the keyboard and select the area of interest, by dragging the cursor. The first selected area will be considered the origin area, while the second area will be the destination. An arrow joining both areas will be drawn, and a a pop-up showing the number of trips that fulfill the criteria will also be displayed. Once again, this query can be refined by properly setting the time filters.

Note that, while in this preliminar prototype we require the use of a keyboard to perform this query, an interactive control is planned to be able to select the starting and ending areas, with the goal of improving the accessibility of the user interface.

7.5 Summary

In this chapter we have presented *Trippy*, a preliminar prototype that takes advantage of the trip representations developed in this thesis to efficiently support queries related to the demand of a public transportation network.

We have proposed an architecture that is flexible enough to extend the current functionality and to include different sources of information in our system, and also

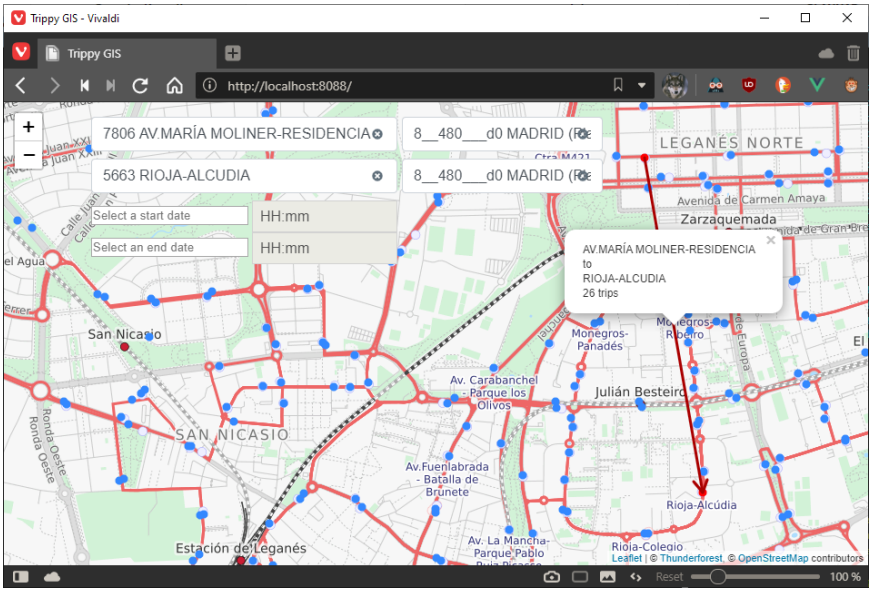


Figure 7.7: Querying for the number of displacements between a starting and an ending stop.

presented a web user-interface to allow any user to easily perform queries of interest to analyze the demand of the underlying transportation system.

Therefore, our *Trippy* prototype permitted us to show that our research on compact data representations to efficiently handle user trips can be interesting for real applications oriented to the support of decision-making processes related to the transportation network administration.

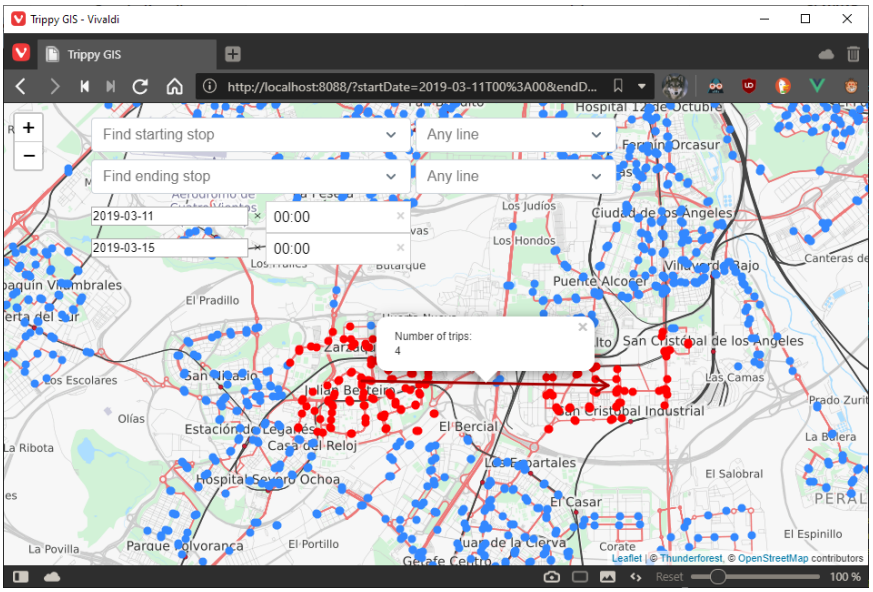


Figure 7.8: Querying for the number of displacements between two areas.

Part III

Thesis summary

Chapter 8

Conclusions and future works

In transportation systems from *smart cities*, new technologies such as traffic monitoring, automatic fare collection (e.g., smartcards) and automatic passenger counting have made possible to generate a huge amount of highly detailed, real-time data useful to define measures that characterize a transportation network. This data is particularly useful because it actually consists of real trips, combining implicitly the demand of the system with either the street network or the service offered by a public transportation system.

At the same time, analyzing these historic records about the demand of the network is within the interest of transportation companies, as well as other entities that may be interested in studying the movements of crowds in an urban context. To make this kind of analysis possible, we have proposed powerful representations based on compact data structures that can easily be adapted for both urban transportation contexts (streets and most public transportation systems), and can handle diverse use cases with considerable memory and time efficiency.

In addition, we have developed a GIS application that integrates the compact representations developed, and includes a simple web-based user interface that makes it possible to analyze the stored information by an end user in a simple way, not requiring them to have a previous knowledge of the underlying technologies. Furthermore, this user interface further validates the choice of an approach based on compact data structures, thus proving that it is possible to develop a competitive product that may be adopted by an interested organization.

Finally, in the context of compact data structures, we believe that this research work may open a new field of practical applications for these structures and algorithms, as have been previously done for the fields of *information retrieval* and *bioinformatics*.

8.1 Contributions

The following list summarizes the main contributions in our work:

1. In the context of trips over urban street networks, we have developed a representation based on a modified CSA and different alternatives of the WT called CTR,

which requires as little as a 36% of the size of an uncompressed baseline, while handling spatio-temporal queries in the order of several microseconds, while offering configurable trade-offs.

2. We have proposed an extensible model for the representation of trips over a public transportation network, that may be adapted for most transportation systems around the world.
3. We have developed two alternative representations for the context of trips over public transportation networks, TTCTR and XCTR, based on the same structures of CTR, which can solve most of our proposed queries about network-aware trip patterns in the order of several mmicroseconds, while needing only about 50% of space compared to a plain (unindexed) representation.
4. We have presented a scheme to compress Summed Area Table without affecting the temporal complexity of its operations, and we applied it in T-Matrices as a structure to accelerate network load queries in the public transportation context.
5. We have developed a GIS prototype including an user interface to analyze the demand of public transportation networks based on our proposed representations, as well as on GIS technologies.

8.2 Future work

While there are many possible future developments for our proposed representations for trips over public transportation networks, we are mostly concerned with finding a single representation that can efficiently handle both kinds of proposed queries in Section 5.1: those focusing in the network load and those focusing in the trip pattern. Finding such a representation is rather challenging, as most solutions that allow to efficiently aggregate multidimensional data (such as stops and times or journeys) cannot support most of our trip pattern queries.

Regarding out *Trippy* prototype, which is still under development, our most urgent goals are to support an intuitive visualization for lines that may enable the user to perform queries over them, as well as the addition of means to query the histogram endpoint, to display the usage of a stop or a line over time with dynamic charts. Finally, this interface may be improved to support different interactive visualizations for the total network usage over all the lines within a time range, in addition to reachability queries, where given a stop in the network we are interested in obtaining the average time taken to arrive to any other stop.

Appendix A

Publications and other research results

Publications

Journals

- Nieves R. Brisaboa, Antonio Fariña, Daniil Galaktionov, and Tirso V. Rodeiro. Trippy: a GIS to visualize aggregated users' trips data built on a compact representation. Draft to submit.
- Nieves R. Brisaboa, Antonio Fariña, Daniil Galaktionov, Tirso V. Rodeiro, and M. Andrea Rodríguez. Improved structures to solve aggregated queries for trips over public transportation networks. Draft to submit.
- Nieves R. Brisaboa, Antonio Fariña, Daniil Galaktionov, and M. Andrea Rodríguez. A Compact Representation for Trips over Networks built on self-indexes. In *Information Systems*, vol 78 (2018), pages 1–22.

International conferences

- Nieves R. Brisaboa, Antonio Fariña, Daniil Galaktionov, Tirso V. Rodeiro, and M. Andrea Rodríguez. New structures to solve aggregated queries for trips over public transportation networks. In *Proceedings of the 25th International Symposium on String Processing and Information Retrieval (SPIRE)* (2018), pages 88–101.
- Nieves R. Brisaboa, Antonio Fariña, Daniil Galaktionov, and M. Andrea Rodríguez. Compact Trip Representation over Networks. In *Proceedings of the 23rd International Symposium on String Processing and Information Retrieval (SPIRE)* (2016), pages 240–253.

International research stays

- *May, 2018 - August, 2018.* Research stay at Universidad de Concepción,

Departamento de Computación (Concepción, Chile).

- *September, 2018 - October, 2018.* Research stay at Universidad de Chile, Departamento de Ciencias de la Computación (Santiago, Chile).

Appendix B

Resumen del trabajo realizado

En este apéndice se presenta un resumen del trabajo realizado durante la tesis. En la sección B.1 se puede encontrar una introducción, donde describimos brevemente el problema a resolver. En la sección B.2 resumimos todas las contribuciones de esta tesis. Finalmente, en la última sección B.3 mencionamos los futuros desarrollos planteados como continuación del trabajo expuesto.

B.1 Introducción

B.1.1 Motivación

En el contexto de las redes de transporte público, los últimos años han visto numerosos avances en las tecnologías inalámbricas, redes de sensores (especialmente aquellas relacionadas con RFID) y computación ubicua, lo que ha llevado a una adopción generalizada de la tecnología de seguimiento de pasajeros por parte de los servicios de transporte público, lo que hace que la recolección de grandes cantidades de datos sobre los hábitos de viaje de estos pasajeros sea más fácil que nunca antes. Esto, a su vez, ha abierto la puerta a la explotación de este tipo de información para estudiar la demanda (uso) de una red, a diferencia de las técnicas conocidas para analizar la oferta (rutas, horarios, etc.). Para estas aplicaciones, no son los datos sobre trayectorias individuales los que tienen importancia, sino las medidas del uso de la red, con las cuales se pueden desarrollar plataformas para el monitoreo del tráfico y las tareas de planificación de carreteras. Ejemplos de medidas útiles son los indicadores de accesibilidad y centralidad, referidos a la facilidad para llegar a determinadas ubicaciones o la importancia de ciertas paradas dentro de una red [MTA07, EGL11, WZTL15]. Todas estas medidas se basan en algún tipo de consultas de conteo que determinan el número de distintos viajes que ocurren dentro de una ventana espacial y/o temporal.

Para habilitar estos nuevos tipos de estudios de demanda, es imperativo desarrollar

mecanismos que hagan posible persistir y administrar eficientemente estas vastas (y siempre crecientes) colecciones de datos. Cuando también tenemos en cuenta que los patrones de consulta eficientes deben ser compatibles para que estos datos sean "útiles", la solución claramente constituye un desafío tecnológico emergente, que se está abordando desde varios dominios diferentes y cientos de soluciones ad-hoc han sido implementados por todas las *Smart Cities* en todo el mundo.

Por consiguiente, una representación práctica para esta información que soporte la indexación eficiente tendría numerosas aplicaciones posibles. En [TCY+18] podemos ver cómo es posible combinar trayectorias GPS con datos de Automated Fare Collection (AFC) para recrear trayectorias completas y estudiar la cantidad de pasajeros por área. Alternativamente, en [WLS+18] las trayectorias completas se infieren de los datos AFC, para luego analizar los patrones de comportamiento y las preferencias de los viajeros con el objetivo de mejorar la eficiencia de la red. Otra aplicación que se habilita mediante dicho análisis es la publicidad dirigida [ZGN+17], ya que los intereses de un usuario pueden ser perfilados por sus patrones de viaje. Otros trabajos se centran en analizar el uso de paradas o estaciones individuales, como [CSC12], donde los autores determinan que los tiempos de congestión en la red de metro de Londres son predecibles y ocurren en intervalos de tiempo estrechos. Armado con dicha información, un usuario puede elegir un patrón de viaje diferente para evitar la multitud y mejorar su experiencia general. Cuando consideramos el transporte público a través de una red de carreteras, podemos encontrar obras centradas en el estudio de los pasajeros de taxis. Un ejemplo notable es [YZZX13], que analiza un sistema de recomendación de taxis bidireccionales, en el que los taxistas señalan los espacios de estacionamiento más rentables mientras los pasajeros son dirigidos a los segmentos de la calle con una alta probabilidad de encontrar un taxi vacante.

Una observación clave de todos los trabajos mencionados anteriormente es que una simple colección de trayectorias o puntos marcados en el tiempo sobre un espacio bidimensional de latitud y longitud no sería lo suficientemente rica como para realizar estos estudios. Por lo tanto, deben trabajar con una representación que considere cierto grado de información *semántica*. Como mínimo, esa información debe incluir referencias a elementos de la red (paradas, líneas o calles) y, a veces, incluso algún identificador de usuario, pudiendo éste último ser anónimo. Por lo tanto, requerimos una representación que difiera de los índices y bases de datos espaciales tradicionales, ya que debe admitir métodos de acceso eficientes basados en elementos de red.

B.1.2 Definición del problema

Considerando la red subyacente, hemos identificado dos contextos distinguibles para las redes de transporte:

- **Redes basadas en calles urbanas**: dentro de estas redes, una trayectoria puede comenzar en cualquier momento y ubicación, y puede seguir cualquier ruta arbitraria de segmentos a lo largo de una red de carreteras definida. Las trayectorias de taxis, bicicletas o flotas de vehículos entran en esta categoría. Para estos sistemas, las consultas de interés pueden involucrar puntos de interés alrededor de los cuales podrían terminar estas trayectorias, o segmentos de carreteras que podrían ser parte de una ruta.

- **Redes de transporte público**: las trayectorias deben comenzar en puntos predefinidos (por lo general, paradas o estaciones) en los horarios establecidos definidos por los vehículos que realizan una parada en esos puntos. Estos vehículos siguen caminos predefinidos a lo largo de estos puntos, formando rutas, y los usuarios que viajan en el mismo vehículo producirían partes idénticas de trayectorias. Esta clasificación se aplica a los sistemas de autobuses y metro, junto con la mayoría de otros sistemas de transporte urbano. Se espera que para una colección de trayectorias de una red de transporte público algunas de las consultas de interés puedan girar en torno a los elementos principales de la red, que son rutas y paradas.

Independientemente del contexto de trabajo, operamos con un modelo de red, que tiende a ser bastante simple en las redes de calles urbanas, consistiendo simplemente en un grafo dirigido con segmentos de calles como nodos, donde la conexión a otros segmentos de calles indica que la navegación es posible. Para las redes de transporte público, podría ser pertinente considerar una representación más rica que un gráfico de paradas y líneas, pero que tenga también en cuenta las rutas formadas por vehículos de transporte, como autobuses o trenes, que visitan las paradas en horarios establecidos en las que los viajeros pueden subir o bajar del vehículo. Un modelo muy conocido que incluye estos elementos de red, entre otros menos interesantes para nuestro problema, es el GTFS,[1] que ha sido ampliamente adoptado por las plataformas de datos abiertos en numerosas *smart cities*.

Para cualquier tipo de red, una *trayectoria* se define como una ruta formada por una secuencia de elementos de red (generalmente paradas o segmentos de calles), que fue recorrida por un solo viajero en un viaje, con un origen y un destino final. En esta definición debemos considerar algunas limitaciones prácticas inherentes a la naturaleza de una trayectoria, ya que se podría argumentar si los viajeros que tardan más de una hora en cambiar una línea a otra realmente las están cambiando, o si simplemente han finalizado sus trayectorias y están comenzando una segunda trayectoria con algún nuevo destino. Estos casos son complicados para decidir inequívocamente en la práctica y, por lo tanto, nuestro enfoque tiende a establecer límites en los tiempos de espera y las distancias del camino entre paradas para una sola trayectoria.

Nuestras definiciones también requieren poder hablar del concepto de tiempo. Cuando trabajamos con un modelo de red de transporte público que integra recorridos formados por vehículos de transporte que siguen líneas, no sería necesario representar la hora exacta en que cada usuario se ha subido a una parada, solo un identificador de recorrido, ya que los tiempos de parada estarían ya disponibles en la red modelada, evitando cierta redundancia en la representación de trayectorias. Alternativamente, para redes de calles urbanas u otros casos donde la información de ruta no está disponible, se puede considerar una representación de intervalos de tiempo para lograr una representación compacta, donde el tiempo se expresaría en intervalos discretos entre uno y treinta minutos.

Existen técnicas de recolección masiva de datos para los dos contextos de red discutidos anteriormente, lo que lleva al problema de manejar eficientemente esta vasta cantidad de información. Por ello, además de las soluciones habituales conocidas de *Big Data*, en nuestro trabajo abrimos una línea de investigación sobre la aplicación de estructuras de datos compactas para los problemas tratados en este ámbito. En particular, es posible aplicar muchas de las técnicas del campo de las estructuras de datos compactas para crear

[1] `https://developers.google.com/transit/gtfs/`

representaciones autoindexadas que admitan patrones de consulta eficientes adaptados a las necesidades de información específicas, al tiempo que ofrece algún tipo de compresión con respecto a una representación más tradicional.

Con nuestras representaciones buscamos una solución que pueda dar una respuesta eficiente a consultas sobre los hábitos de movilidad de los usuarios de un sistema de transporte. Las consultas a considerar pueden variar en función del contexto de la red de transporte. Así, para un contexto de redes de calles, como el de los movimientos en taxi, podríamos destacar las siguientes consultas:

- Número medio de movimientos que han pasado por una calle o tramo de calle durante un intervalo temporal.
- Número de viajes que se han realizado a partir de un origen determinado durante los fines de semana.
- De aquellos viajes que han empezado en un origen, obtener cuántos han terminado el viaje en un destino concreto.
- Par cada día, obtener las diez calles donde más frecuentemente comienzan los viajes.

Por otro lado, en el contexto de las redes de transporte público, nos pueden interesar consultas que tengan en consideración los elementos de la red, como pueden ser líneas/rutas y los recorridos individuales de éstas. Por tanto, algunas de las consultas relevantes podrían ser:

- Número de usuarios que se han subido a las paradas extremas de una línea durante las horas puntas de la mañana.
- Carga media del vehículo de transporte (tren o autobús) entre dos paradas de una línea.
- Número de veces que una parada concreta se ha utilizado para cambiar de línea, en lugar de como un destino final.
- Número de viajes originados desde una parada concreta de una línea que hayan terminado en otra parada concreta dentro de un intervalo temporal.

Es importante destacar de estas consultas que, a pesar de que muchas de ellas se refieran a calles o a paradas concretas, es habitual que la red subyacente no permanezca estática durante todo el tiempo que comprende el conjunto de datos, sino que existan cambios de naturaleza temporal (obras o accidentes) o permanente (paradas o líneas cerradas o abiertas). Sin embargo, no consideramos que estos cambios en la red sean importantes para las tareas finales, ya que el efecto en la demanda de la movilidad es mínimo: cuando una parada de metro deja de estar disponible, los usuarios simplemente utilizarán otra parada cercana, de forma que se sigue pudiendo realizar un análisis de movilidad sobre zonas.

Por último, hemos considerado que una solución usable también requeriría disponer de una herramienta que provea una interfaz de usuario simple que permita la explotación de esta información por parte de investigadores, empresas de transporte, administraciones municipales y cualquier otro tipo de usuarios finales. Esta interfaz debe, como mínimo, permitir visualizar los elementos de la red en un mapa, además de permitir la capacidad de realizar consultas sobre estos elementos de una manera intuitiva y accesible, respetando los principios de calidad habituales de cualquier software similar orientado a usuario.

B.2 Contribuciones y conclusiones

En los sistemas de transporte de las *smart cities*, las nuevas tecnologías como el monitoreo del tráfico, la existencia de mecanismos de pago automatizados (por ejemplo, tarjetas inteligentes) y el conteo automático de pasajeros han permitido generar una gran cantidad de información altamente detallada, datos en tiempo real útiles para definir medidas que caracterizan una red de transporte. Estos datos resultan particularmente interesantes porque consisten en viajes reales, combinando así implícitamente la demanda del sistema con la red de calles o el servicio ofrecido por un sistema de transporte público.

Al mismo tiempo, el análisis de estos registros históricos sobre la demanda de la red puede interesar a las compañías de transporte, así como a otras entidades que puedan estar interesadas en estudiar los movimientos de los viajeros en un contexto urbano. Para hacer posible este tipo de análisis, hemos propuesto representaciones eficaces y flexibles basadas en estructuras de datos compactas que pueden adaptarse fácilmente a contextos de transporte urbano (calles y la mayoría de los sistemas de transporte público), y que pueden manejar diversos casos de uso con considerable eficiencia temporal y de memoria.

La representación propuesta para las redes de calles, llamada CTR, codifica los identificadores de los nodos de red que se recorren en forma de una cadena de texto, al igual que la secuencia de los intervalos de tiempo en los que se visita cada uno de estos nodos. Con esta codificación hemos construido un CSA para las secuencias de nodos (componente espacial) y también ofrecemos dos variantes del WT (el HTWT y el WM) como alternativas para las secuencias de intervalos de tiempo (componente temporal). Ambas estructuras pueden utilizarse de forma conjunta para resolver consultas espacio-temporales complejas relacionadas con la utilización de la red.

En cuanto a las redes de transporte público, hemos definido un modelo que puede usarse para representar la demanda de la mayoría de los sistemas de transporte existentes, con el cuál toda trayectoria queda representada como una secuencia de subidas y bajadas en varios puntos de la red. Para cada subida es necesario definir: la parada, la línea y el recorrido concreto del vehículo de transporte. Ofrecemos dos alternativas para la representación de estos componentes: el TTCTR, que codifica las combinaciones válidas de parada y línea en un mismo vocabulario con el que se construye un CSA; y el XCTR, donde la secuencia de líneas se mantiene en una WM separado. Ambas representaciones usan una WM adicional para los códigos de recorrido, que se pueden usar para filtrar resultados temporalmente al disponer de la hora de inicio de cada recorrido y el tiempo medio de llegada a cada una de las paradas de la línea.

Además, hemos desarrollado un prototipo de aplicación GIS (llamada *Trippy* que integra las representaciones compactas desarrolladas, lo que permite analizar la información almacenada por un usuario final de una manera simple, sin requerir que tengan un conocimiento previo de las tecnologías subyacentes. Además, esta interfaz de usuario valida aún más la elección de un enfoque basado en estructuras de datos compactas, demostrando así que es posible desarrollar un producto competitivo y ágil que pueda ser adoptado por una organización.

Finalmente, en el contexto de la investigación de estructuras de datos compactas, creemos que este trabajo puede abrir un nuevo campo de aplicaciones prácticas para estas estructuras y algoritmos, como se ha hecho anteriormente para los campos de *recuperación de información* y *bioinformática*.

La siguiente lista resume las principales contribuciones en nuestro trabajo:

1. En el contexto de viajes a través de redes de calles urbanas, hemos desarrollado una representación basada en un CSA modificado y diferentes alternativas del WT llamado CTR, que requiere tan solo un 36% del tamaño de una línea base sin comprimir, al tiempo que maneja consultas espacio-temporales en el orden de varios microsegundos, al tiempo que ofrece trade-offs configurables.
2. Hemos propuesto un modelo extensible para la representación de viajes a través de una red de transporte público, que puede adaptarse para la mayoría de los sistemas de transporte del mundo.
3. Hemos desarrollado dos representaciones alternativas para el contexto de viajes a través de redes de transporte público, TTCTR y XCTR, basadas en las mismas estructuras del CTR, y que pueden resolver la mayoría de nuestras consultas propuestas acerca de patrones de trayectorias sobre la red en el orden de varios microsegundos, mientras que solo se necesita alrededor del 50% de espacio del que requeriría una representación tradicional sin índice.
4. Hemos presentado un método para comprimir Summed Area Table (SAT) sin afectar a la complejidad temporal de sus operaciones, y lo aplicamos en T-Matrices como una estructura para acelerar las consultas de carga de red en el contexto del transporte público.
5. Hemos desarrollado un prototipo preliminar de aplicación para analizar la demanda de redes de transporte público que utiliza nuestras representaciones propuestas, así como tecnologías GIS.

B.3 Trabajo futuro

Si bien consideramos que hay muchos desarrollos futuros posibles para las representaciones que hemos propuesto para viajes a través de redes de transporte público, nos preocupa principalmente encontrar una representación única que pueda manejar eficientemente los dos tipos de consultas propuestas en la sección 5.1: las consultas sobre carga de la red y también aquellas sobre los patrones de viaje. Encontrar tal representación resulta considerablemente complejo, ya que la mayoría de las soluciones que permiten agregar eficientemente datos multidimensionales (como paradas y horarios o viajes) no pueden soportar la mayoría de nuestras consultas de patrones de viaje.

En cuanto a los métodos de evaluación de nuestras representaciones para las trayectorias en redes de transporte público, consideramos importante poder disponer de registros de trayectos de transporte reales, con los que podríamos realizar la evaluación sin tener que depender de la generación de trayectos sintéticos sobre las redes. Estos registros podrían usarse también para aumentar artificialmente nuestro conjunto de datos, ya que sería posible repetir los mismos trayectos sobre días diferentes, con o sin pequeñas modificaciones, o también utilizar esa información para generar nuevos trayectos. Actualmente es posible recolectar la información necesaria para reconstruir estos trayectos, como por ejemplo con el uso de las tarjetas de transporte tanto a la entrada como a la salida de las estaciones de metro, o con aplicaciones móviles para los pasajeros de autobús.

Por otro lado, en relación a nuestro prototipo desarrollado, *Trippy*, nuestros objetivos más urgentes son soportar una visualización intuitiva de líneas que permita al usuario realizar consultas sobre ellas, así como la adición de medios para consultar el *endpoint del histograma*, para mostrar el uso de una parada o una línea a lo largo del tiempo con gráficos dinámicos. Finalmente, la interfaz de usuario actual se puede mejorar para admitir diferentes visualizaciones interactivas para el uso total de la red en todas las líneas dentro de un rango de tiempo. Además podríamos incluir consultas de accesibilidad, donde a partir e una parada de la red estamos interesados en obtener el tiempo promedio necesario para llegar a cualquier otra parada.

Bibliography

[AAMF16] Azalden Alsger, Behrang Assemi, Mahmoud Mesbah, and Luis Ferreira. Validating and improving public transport origin–destination estimation algorithm using smart card fare data. *Transportation Research Part C: Emerging Technologies*, 68:490–506, 2016.

[ADWK+17] Shamal Al-Dohuki, Yingyu Wu, Farah Kamw, Jing Yang, Xin Li, Ye Zhao, Xinyue Ye, Wei Chen, Chao Ma, and Fei Wang. Semantictraj: A new approach to interacting with massive taxi trajectories. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):11–20, 2017.

[BC+15] Ashish Bhaskar, Edward Chung, et al. Passenger segmentation using smart card data. *IEEE Transactions on Intelligent Transportation Systems*, 16(3):1537–1548, 2015.

[BFG+18] Nieves R Brisaboa, Antonio Fariña, Daniil Galaktionov, Tirso V Rodeiro, and M Andrea Rodríguez. New structures to solve aggregated queries for trips over public transportation networks. In *Proc. 25th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 11147, pages 88–101. Springer, 2018.

[BGBNP19] Nieves R Brisaboa, Adrián Gómez-Brandón, Gonzalo Navarro, and José R Paramá. Gract: A grammar-based compressed index for trajectory data. *Information Sciences*, 2019.

[BLN09] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. k2-trees for compact web graph representation. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5721, pages 18–30. Springer, 2009.

[BN09] Jérémy Barbay and Gonzalo Navarro. Compressed representations of permutations, and applications. *arXiv preprint arXiv:0902.1038*, 2009.

[BN13] Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013.

[Boe17] Geoff Boeing. Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems*, 65:126–139, 2017.

[BPSW05] Sotiris Brakatsoulas, Dieter Pfoser, Randall Salas, and Carola Wenk. On map-matching vehicle tracking data. In *Proc. 31st International Conference on Very Large Data Bases (VLDB)*, pages 853–864. VLDB Endowment, 2005.

[CEP03] V. Prasad Chakka, Adam Everspaugh, and Jignesh M. Patel. Indexing large trajectory data sets with SETI. In *Proc. 1st Conference on Innovative Data Systems Research (CIDR)*, pages 1–12, 2003.

[CFMPN16] Francisco Claude, Antonio Farina, Miguel A Martínez-Prieto, and Gonzalo Navarro. Universal indexes for highly repetitive document collections. *Information Systems*, 61:1–23, 2016.

[CG18] Bertil Chapuis and Benoit Garbinato. Geodabs: Trajectory indexing meets fingerprinting at scale. *arXiv preprint arXiv:1803.04292*, 2018.

[CN08] Francisco Claude and Gonzalo Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187, 2008.

[CNO15] Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez . The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.

[Cro84] Franklin C Crow. Summed-area tables for texture mapping. In *ACM SIGGRAPH computer graphics*, volume 18, pages 207–212. ACM, 1984.

[CSC12] Irina Ceapa, Chris Smith, and Licia Capra. Avoiding the crowds: understanding tube station congestion patterns from trip data. In *Proc. of the ACM SIGKDD international workshop on urban computing*, pages 134–141. ACM, 2012.

[CSU10] Jae-Woo Chang, Myoung-Seon Song, and Jung-Ho Um. TMN-tree: new trajectory index structure for moving objects in spatial networks. In *Proc. 10th International Conference on Computer and Information Technology (CIT)*, pages 1633–1638, 2010.

[CT06] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2nd edition, 2006.

[dAG05] Victor Teixeira de Almeida and Ralf Hartmut Güting. Indexing the Trajectories of Moving Objects in Networks. *GeoInformatica*, 9(1):33–60, 2005.

[DCG+18] Xin Ding, Lu Chen, Yunjun Gao, Christian S Jensen, and Hujun Bao. Ultraman: a unified platform for big trajectory data management and analytics. *Proceedings of the VLDB Endowment*, 11(7):787–799, 2018.

[DIGV15] Maria Luisa Damiani, Hamza Issa, Ralf Hartmut Güting, and Fabio Valdés. Symbolic trajectories and application challenges. *SIGSPATIAL Special*, 7(1):51–58, 2015.

[DYGL15] Zhiming Ding, Bin Yang, Ralf Hartmut Güting, and Yaguang Li. Network-matched trajectory-based moving-object database: Models and applications. *IEEE Transactions on Intelligent Transportation Systems*, 16(4):1918–1928, 2015.

[EGL11] Ahmed El-Geneidy and David Levinson. Place rank: Valuing spatial interactions. *Networks and Spatial Economics*, 11(4):643–659, 2011.

[EGSV99] Martin Erwig, Ralf Hartmut Güting, Markus Schneider, and Michalis Vazirgiannis. Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. *GeoInformatica*, 3(3):269–296, 1999.

[FBN+12] Antonio Fariña, Nieves R. Brisaboa, Gonzalo Navarro, Francisco Claude, Ángeles S. Places, and Eduardo Rodríguez. Word-based Self-Indexes for Natural Language Text. *ACM Transactions on Information Systems (TOIS)*, 30(1):article 1:, 2012.

[FGM+16] Antonio Fariña, Travis Gagie, Giovanni Manzini, Gonzalo Navarro, and Alberto Ordóñez. Efficient and compact representations of some non-canonical prefix-free codes. In *Proc. 23rd International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 9954, pages 50–60. Springer, 2016.

[FGNS00] Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider. A data model and data structures for moving objects databases. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 319–330, 2000.

[FGNV09] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics*, 13:1–12, 2009.

[FM00] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

[FMPC+19] Antonio Fariña, Miguel A Martínez-Prieto, Francisco Claude, Gonzalo Navarro, Juan J Lastra-Díaz, Nicola Prezza, and Diego Seco. On the reproducibility of experiments of indexing repetitive document collections. *Information Systems*, 83:181–194, 2019.

[Fre03] Elias Frentzos. Indexing Objects Moving on Fixed Networks. In *Proc. 8th International Symposium on Spatial and Temporal Databases (SSTD)*, pages 289–305, 2003.

[GBE+00] Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. A foundation for representing and querying moving objects. *ACM Transactions on Database Systems*, 25(1):1–42, 2000.

[GBE+03] Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos Lorentzos, Enrico Nardelli, Markus Schneider, and Jose R. R. Viqueira. Chapter 4: Spatio-temporal models and languages: An approach based on data types. In *Spatio-Temporal Databases: The CHOROCHRONOS Approach*, LNCS 2520, pages 117–176, 2003.

[GdAD06] Ralf Hartmut Güting, Teixeira de Almeida, and Zhiming Ding. Modeling and querying moving objects in networks. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 15(2):165–190, 2006.

[GGV03] Roberto Grossi, Alexander Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[GM59] Edgar N Gilbert and Edward F Moore. Variable-length binary encodings. *Bell System Technical Journal*, 38(4):933–967, 1959.

[GNP12] Travis Gagie, Gonzalo Navarro, and Simon J Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426:25–41, 2012.

[GS05] Ralf Hartmut Güting and Markus Schneider. *Moving Objects Databases*. Morgan Kaufmann, 2005.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 47–57, 1984.

[GV00] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd ACM Symposium on Theory of Computing (STOC)*, pages 397–406, 2000.

[HA03] John E Harmon and Steven J Anderson. *The design and implementation of geographic information systems*. John Wiley & Sons, 2003.

[Hor77] Yasuichi Horibe. An improved bound for weight-balanced tree. *Information and Control*, 34(2):148–151, 1977.

[HT71] Te C. Hu and Alan C. Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.

[Huf52] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[Jac89] Guy Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[JL09] Bin Jiang and Chengke Liu. Street-based topological representations and analyses for predicting traffic flow in gis. *International Journal of Geographical Information Science*, 23(9):1119–1137, 2009.

[JSR17] A John, M Sugumaran, and RS Rajesh. Performance analysis of the past, present and future indexing methods for spatio-temporal data. In *Proc. of 2nd International Conference on Communication and Electronics Systems (ICCES)*, pages 645–649. IEEE, 2017.

[Kar11] Charles FF Karney. Transverse mercator with an accuracy of a few nanometers. *Journal of Geodesy*, 85(8):475–485, 2011.

[KPTT14] Benjamin Krogh, Nikos Pelekis, Yannis Theodoridis, and Kristian Torp. Path-based queries on trajectory data. In *Proc. 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, pages 341–350, 2014.

[KTXI18] Satoshi Koide, Yukihiro Tadokoro, Chuan Xiao, and Yoshiharu Ishikawa. Cinct: Compression and retrieval for massive vehicular trajectories via relative movement labeling. In *Proc. 34th International Conference on Data Engineering (ICDE)*, pages 1097–1108. IEEE, 2018.

[LGMR15] Paul A Longley, Michael F Goodchild, David J Maguire, and David W Rhind. *Geographic Information Science and Systems*. John Wiley & Sons, 2015.

[LKG+12] Yu Liu, Chaogui Kang, Song Gao, Yu Xiao, and Yuan Tian. Understanding intra-urban trip patterns from taxi trajectory data. *Journal of Geographical Systems*, 14(4):463–483, 2012.

[LM00] N Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[LS07] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, November 2007.

[LWW+17] Yongxin Liu, Xiaoxiong Weng, Jiafu Wan, Xuejun Yue, Houbing Song, and Athanasios V Vasilakos. Exploring data validity in transportation systems for smart cities. *IEEE Communications Magazine*, 55(5):26–33, 2017.

[MM93] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[MMGF+13] Luis Moreira-Matias, Joao Gama, Michel Ferreira, Joao Mendes-Moreira, and Luis Damas. Predicting taxi–passenger demand using streaming data. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1393–1402, 2013.

[MP17] Alistair Moffat and Matthias Petri. ANS-based index compression. In *Proc. of the 2017 ACM on Conference on Information and Knowledge Management*, pages 677–686. ACM, 2017.

[MPPP19] Oliver Magiera, Rosa Pink, David Piper, and Christopher Poeplau. Sacabench: Benchmarking suffix array construction. In *Proc. 26th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume LNCS 11811, pages 407–416. Springer, 2019.

[MTA07] Catherine Morency, Martin TrÃľpanier, and Bruno Agard. Measuring transit use variability with smart-card data. *Transport Policy*, 14(3):193–203, 2007.

[Mun96] Ian Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

[MW14] Xiaolei Ma and Yinhai Wang. Development of a data-driven platform for transit performance measures using smart card and gps data. *Journal of Transportation Engineering*, 140(12):04014063, 2014.

[Nav16] Gonzalo Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016.

[NM07] Gonzalo Navarro and Veli Mäkinen. Compressed Full-text Indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[NS98] Mario A. Nascimento and Jefferson R.O. Silva. Towards historical R-trees. In *Proc. ACM symposium on Applied Computing (SAC)*, pages 235–240. ACM, 1998.

[NZC11] Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011.

[Ord16] Alberto Ordóñez. *Statistical and repetition-based compressed data structures.* PhD thesis, Department of Computer Science, University of A Coruña, 2016.

[OS07] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.

[PJT00] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel Approaches in Query Processing for Moving Object Trajectories. In *Proc. 26th International Conference on Very Large Data Bases (VLDB)*, pages 395–406, 2000.

[PT14] Nikos Pelekis and Yannis Theodoridis. *Mobility Data Management and Exploration.* Springer, 2014.

[RRR02] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[RRS18] Rodrigo Rivera, M Andrea Rodríguez, and Diego Seco. Faster and smaller two-level index for network-based trajectories. In *Proc. 25th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 11147, pages 348–362. Springer, 2018.

[Sad03] Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[Sha48] Claude Elwood Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423, 1948.

[Spa01] Stefano Spaccapietra. Editorial: Spatio-Temporal Data Models and Languages. *GeoInformatica*, 5(1):5–9, 2001.

[SPZO+11] Iulian Sandu Popa, Karine Zeitouni, Vincent Oria, Dominique Barth, and Sandrine Vial. Indexing In-network Trajectory Flows. *The VLDB Journal*, 20(5):643–669, 2011.

[SWCD97] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Modeling and querying moving objects. In *Proc. 13th International Conference on Data Engineering (ICDE)*, pages 422–432, 1997.

[TCMBR14] Sui Tao, Jonathan Corcoran, Iderlina Mateo-Babiano, and David Rohde. Exploring bus rapid transit passenger travel behaviour using big data. *Applied geography*, 53:90–104, 2014.

[TCY+18] Wei Tu, Rui Cao, Yang Yue, Baoding Zhou, Qiuping Li, and Qingquan Li. Spatial variations in urban public ridership derived from gps trajectories and smart card data. *Journal of Transport Geography*, 69:45–57, 2018.

[TP01] Yufei Tao and Dimitris Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *Proc. 27th International Conference on Very Large Data Bases (VLDB)*, pages 431–440, 2001.

[Wan11] Wenpeng Wang. Review on hybrid flow shop scheduling. In *Proc. of the 14th International Conference of Information Technology, Computer Engineering and Management Sciences*, volume 4, pages 7–10. IEEE, 2011.

[WLL14] WL Wang, SM Lo, and SB Liu. Aggregated metro trip patterns in urban areas of hong kong: Evidence from automatic fare collection records. *Journal of Urban Planning and Development*, 141(3), 2014.

[WLS+18] Xiaoxiong Weng, Yongxin Liu, Houbing Song, Shushen Yao, and Pengfei Zhang. Mining urban passengers' travel patterns from incomplete data with use cases. *Computer Networks*, 134:116–126, 2018.

[WXCJ98] Ouri Wolfson, Bo Xu, Sam Chamberlain, and Liqin Jiang. Moving objects databases: Issues and solutions. In *Proc. 10th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 111–122, 1998.

[WZTL15] Guihua Wang, Yuanguang Zhong, Chung-Piaw Teo, and Qizhang Liu. Flow-based accessibility measurement: The place rank approach. *Transportation Research Part C: Emerging Technologies*, 56:335–345, 2015.

[Xia15] Ningchuan Xiao. *GIS algorithms*. Sage, 2015.

[YZZX13] Nicholas Jing Yuan, Yu Zheng, Liuhang Zhang, and Xing Xie. T-finder: A recommender system for finding passengers and vacant taxis. *IEEE Transactions on knowledge and data engineering*, 25(10):2390–2403, 2013.

[ZGN+17] Dongxiang Zhang, Long Guo, Liqiang Nie, Jie Shao, Sai Wu, and Heng Tao Shen. Targeted advertising in public transportation systems with quantitative evaluation. *ACM Transactions on Information Systems (TOIS)*, 35(3):20, 2017.

[ZTG+11] John Zimmerman, Anthony Tomasic, Charles Garrod, Daisy Yoo, Chaya Hiruncharoenvate, Rafae Aziz, Nikhil Ravi Thiruvengadam, Yun Huang, and Aaron Steinfeld. Field trial of tiramisu: crowd-sourcing bus arrival times to spur co-design. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1677–1686. ACM, 2011.