Lossless compression of public transit schedules

Sebastian Wandelt, Xiaoqian Sun* and Yanbo Zhu*

***Abstract*—Transit agencies electronically publish transit schedules and route data to improve the customer experience or as part of the open data initiative. The release of such data poses several data management challenges, because schedules for a single city can easily exceed storage requirements of several hundreds of megabyte. One way to deal with these challenges is data compression. The encoding of public transit schedules often follows the General Transit Feed Specification (GTFS), which cannot be compressed well out of the box.**

We propose GTFSCompress, an algorithm for compression of GTFS data, based on referential compression queues, which compress a column stream depending on previously seen items in this stream and other streams. Our evaluation on ten real-world datasets shows that GTFS feeds can be compressed by a factor of 100 and more. This is up to one order of magnitude better than using the best standard compressors.

***Index Terms*—Traffic information systems, Data management**

I. Introduction

In the last decades, urban travel patterns undergo tremendous changes. While the amount of automobile usage saw an unprecedented growth, recent trends indicate an increased importance of public transportation infrastructures [1]. Public transportation helps to address the following problems, among others: High traffic congestion [2], excessive energy consumption [3], high CO2 emission [4], and increased awareness of public health [5]. Furthermore, the extraordinary growth in excessive smartphone usage has led to new scenarios for guiding citizens through multi-modal urban environments [6].

Recently, transit agencies have started to electronically publish transit schedules and route data to improve the customer experience [7]. Transit data is often released in a standardized, broadly accepted format called General Transit Feed Specification (GTFS) [8]. This feed format was originally developed by Google in cooperation with Portland TriMet. Since then, GTFS feeds have found their ways into broad usage within transport community. We describe some of these GTFS-based approaches next; for a complete review see [7].

[9] describe cooperative transit tracking with GTFS feeds, a crowd-sourced alternative to official transit tracking. [10] develop a travel assistance device to aid transit riders with special needs; public transit information is obtained by GTFS feeds. [11] characterize transit service quality with excess journey time (arrival delay); public transportation schedules for London Overground are obtained from GTFS feeds. [12] develop a system to extract the personal transit travel diaries, by matching GPS traces to vehicle locations and public transportation schedules from GTFS feeds. The system is evaluated in San Francisco's Muni network. [13] analyze aggregate commute mode shares; particularly focusing on the share of transit relative to auto. The authors' calculations are accomplished using only publicly-available data sources, one of them is a GTFS feed of Metro Transit from Twin Cities. Web-based systems for reporting positions of public transit vehicles have been developed based on GTFS data, for instance, OneBusAway [14] and TRAVIC [15].

One major limitation of GTFS data is that the storage requirement is very high. Managing the public traffic schedules for Paris, for instance, needs 700 MB of storage. Standard compression tools cannot compress GTFS data efficiently, since they cannot exploit the structure inherent to the specification of GTFS; frequently incurring context changes also make compression hard. Therefore, the design and development of efficient compression tools are one requirement for dealing with big data in transportation. Recent data handling techniques for GTFS files are either based on lossy compression [16] or focus on uncompressed indexing [17]. Standard lossless compression techniques, on the other hand, do not fully exploit existing compression potential.

In this paper, we present GTFSCompress, an algorithm for compression of GTFS data. Given that GTFS datasets are essentially a set of relations, we propose to extend a traversal strategy from the database domain, so-called column-traversal. The processing of columns separately leads to so-called *vertical compression* [18], which achieves higher compression ratios than standard horizontal compression. Based on the idea of column-wise compression, we propose a new data structure called *referential compression queues*, which compress a column stream depending on previously seen items in this stream and other streams. Our evaluation on ten real-world GTFS datasets shows that the compression ratio is increased by a factor of 5–10, compared to standard compression tools. The major contributions of our work are:

1) Compared to standard compression algorithms, we propose a column-wise traversal strategy, which avoids frequent context changes, since compressing mixed content is much harder than compressing homogeneous values.
2) Opposed to [16], we propose a lossless compression technique, which means that the original file can be reconstructed during decompression. This is important for downstream analysis, especially when dealing with geospatial data. Moreover, we show that even lossless compression can yield very high compression ratios. Therefore, we do not need to loose information to compress GTFS files with high compression ratios.
3) Opposed to [17], we focus on compressed representations and lossless reconstruction, instead of retrieval/query performance. Storing GTFS relations together with indexes increases the required amount of storage significantly.
4) Our compression technique can be seen as a very general way for compressing CSV files; not only applicable to GTFS, but also other relational data. Furthermore, our algorithms can be seamlessly applied, in case the structure of GTFS files is revised, as long as the underlying data model is still relational.

S. Wandelt is with Beihang University, Beijing, China e-mail: (wandelt@informatik.hu-berlin.de).

X. Sun is with School of Electronic and Information Engineering, Beihang University, Beijing, China e-mail: (sunxq@buaa.edu.cn).

Y. Zhu is with Aviation Data Communication Corporation, No. 238 Baiyan Building, 100191 Beijing, China e-mail: (zyb@adcc.com.cn).

* Shared corresponding authors: X. Sun and Y. Zhu.

The remaining part of this paper is structured as follows. Section II discusses related work on compression of GTFS feeds and the GTFS format is reviewed in Section III. We introduce our GTFS compression technique and its implementation in Section IV. Section V evaluates our new techniques on ten real-world GTFS datasets. The paper is concluded in Section VI.

II. Related Work

A large body of literature has discussed how to use GTFS feeds for increasing travelers' experiences during public transit (see our summary above). There is, however, only few work dealing with the associated data management challenges. [16] develop a data structure for storing GTFS feeds on mobile devices. Since their work is closely related to ours, we discuss the major limitations in detail.

1) **The compression method is lossy.** This means that upon compression, some data is inevitably lost and cannot be recovered during decompression. We provide only a few examples here: The authors omit some parts of GTFS feeds completely (e.g. agencies, frequencies, and custom sub-feeds). The authors make several decisions about what parts of the feed are relevant and up to which degree; whether these parts are indeed relevant, clearly depends on the context and use case. It should be noted that the goal of [16] was not to compress GTFS data in general, but to create an implementation of a commercial public transit application using different data sources (including GTFS feeds).
2) **The compression method possibly requires human interaction for new cities.** The compression of GTFS files needs human interaction at several stages, such that an automatic compressor/decompressor for GTFS feeds cannot be implemented directly[1]. We provide some examples here: Routes in the GTFS file are grouped by hand *according to various features which depend on the city*. Moreover, stops/shapes which are *not more than a few meters away from each other*, are merged (in few cases).
3) **The formal description is sometimes incomplete.** Throughout the paper, the authors leave details about their implementations open. For instance, the maximum distance between merged stops and the degree of similarity between merged routes are not sufficiently described, which makes the reproduction of their results difficult. The code-snippets supplied in the Appendix of [16] do not help to resolve these issues, since they are enumerating some C++ structures without stating how they are used/populated. The complete code for compressing GTFS feeds is not available either.

Therefore, while the paper raises interesting ideas for compression, its impact on the transportation community is rather limited. Even after careful analysis of the paper, it is still unclear, how much of the original feed is left exactly in their *compressed* representation. This is problematic, since compression algorithms are, in general, intended to be lossless.

[17] present a web-based archive for transit performance data. The authors, however, do not address data storage challenges, but data integration issues, for instance, merging GTFS feeds with other transit (live) data. Similarly, the Portal transportation data archive[2] is described in [19]. Within 10 years, the Portal has grown to approx. 3 TB of transportation-related data, covering, for instance, freeway data, arterial signal data, and travel time. While the authors describe use cases for the Portal, storage management challenges are not addressed.

GTFS was proposed quite recently and its use (in academic research) is spurred lately by an increasing publication of real-world transportation datasets. Currently, studies on urban transportation often deal with small/single urban areas only, where the big data problem is not so evident, since, for instance, even all the transportation data for Paris can be managed on a commodity laptop nowadays. However, for future research, we envision the publication of larger GTFS feeds for a) huge metropolitan areas and b) different temporal periods. Particularly, the latter item will play a key role for analysis of dynamics in urban transportation. At this time, storing and retrieving large sets of (worldwide) GTFS feeds will become a challenging problem, which will also be reflected by publications in this area.

Research on general data compression algorithms can be broken down into two areas:

1) **Universal compression schemes:** Such algorithms compress any kind of input, without exploitation of the file structure or content. Example for these algorithms are LZ77 [20], PPM [21], and BWT [22]. These algorithms have in common that they work on the symbol level (bits/bytes) and maintain frequency statistics during compression, where frequently occurring symbols receive shorter codes than rare symbols.
2) **Domain-dependent compression schemes:** Such algorithms exploit domain information to significantly increase compression rates, compared to universal compression schemes. For instance, biological sequences are compressed using dictionary-based, statistical, or referential encoding, the latter ones taking a single genome as a reference and encoding other genomes of the same species by denoting the deviations only [23]. Similarly, video sequences are often encoded frame by frame, where only deviations from a to-be-compressed frame and a fixed key frame are encoded [24]. Other domain-specific compression methods have been developed, among others, for GPS traces [25], aircraft trajectories [26], weather radar data [27], and sensor networks [28].

III. General Transit Feed Specification

The General Transit Feed Specification (GTFS) [8] was originally developed by Google, in cooperation with Portland TriMet[3], as a standardized way to model transit schedules. Below, we describe the files in a GTFS feed in detail. Each file describes information in the relational data model and is serialized using comma-separated value files. Please note that the feeds sometimes contain additional files (not modeled in the specification).

- **agency.txt**: A list of transit agencies providing data.
- **stops.txt**: All stops of the feed for pick up or drop off.
- **trips.txt**: All trips in the feed, where a trip is a sequence of stops which occur at specific time.
- **routes.txt**: All routes in the feed, where a route is a group of distinct trips.
- **stop_times.txt**: Arrival and departure time of vehicles at individual stops for trips.
- **calendar.txt**: Dates for services using a weekly schedule.

[1] According to the authors, most of these hand-made changes either 1) increase the user experience or 2) repair inconsistencies in the GTFS dataset.

[2] http://portal.its.pdx.edu/

[3] http://trimet.org/

- **calendar_dates.txt**: Exceptions for calendar.txt file.
- **fare_attributes.txt**: Information on fares.
- **fare_rules.txt**: Information on how to apply fares.
- **shapes.txt**: Information on visualization of routes.
- **frequencies.txt**: Headway (time between trips) for routes with variable frequency of service.
- **transfers.txt**: Information about transfer points.
- **feed_info.txt**: Information about the feed.

IV. Compression of GTFS

In this section we present our algorithms for compressing GTFS data. The algorithms are implemented in our tool GTFSCompress. The formal compression model is introduced in Section IV-A, and the actual implementation of GTFSCompress is presented in Section IV-B.

A. Formal compression model

A *string* s is a finite sequence of characters from an alphabet Σ. The concatenation of two strings s and t is denoted with $s \circ t$. A string s is a *substring* of string t, if there exist two strings u and v (possibly of length 0), such that $t = u \circ s \circ v$. The length of a string s is denoted with $|s|$ and the substring starting at position i with length n is denoted with $s(i, n)$. $s(i)$ is an abbreviation for $s(i, 1)$. All positions in a string are zero-based, i.e., the first character is accessed by $s(0)$.

Definition IV.1 (GTFSRelation)**.** *A* GTFSRelation *with m columns and n rows is a matrix*

$$M_{GTFS} = \begin{array}{c} \\ 1 \\ 2 \\ \dots \\ n \end{array} \begin{array}{c} \begin{array}{cccccc} h_1 & h_2 & h_3 & \dots & h_{m-1} & h_m \end{array} \\ \begin{bmatrix} x_{1,1} & x_{2,1} & x_{3,1} & \dots & x_{m-1,1} & x_{m,1} \\ x_{1,2} & x_{2,2} & x_{3,2} & \dots & x_{m-1,2} & x_{m,2} \\ \dots & \dots & \dots & \dots & \dots & \\ x_{1,n} & x_{2,n} & x_{3,n} & \dots & x_{m-1,n} & x_{m,n} \end{bmatrix} \end{array}$$

The column of M_{GTFS} with header h_i is denoted with $M^{h_i}_{GTFS} = [x_{i,1}, x_{i,2}, x_{i,3}, \dots, x_{i,n-1}, x_{i,n}]$. The number of columns is denoted with $colCount(M_{GTFS}) = m$ and the number of rows is denoted with $rowCount(M_{GTFS}) = n$. The item at column i and row j, i.e. $x_{i,j}$, is denoted with $M_{GTFS}[i, j]$.

Example 1. *We discuss an example of a valid GTFS relation, for the type stop_times.txt. M_{EX1} provides a simple description of two trips ($trip\_id = \{1, 2\}$) from stop 9230999 to stop 9220070. The second trip ($trip\_id = 2$) starts 10 minutes after the first trip.*

$$M_{EX1} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cccccc} \textit{trip\_id} & \textit{arrival\_time} & \textit{departure\_time} & \textit{stop\_id} & \textit{stop\_sequence} & \textit{pickup\_type} \end{array} \\ \begin{bmatrix} 1 & 04:45:00 & 04:45:30 & 9230999 & 1 & 0 \\ 1 & 04:51:00 & 04:51:30 & 9230400 & 2 & 1 \\ 1 & 04:59:00 & 04:59:30 & 9220019 & 3 & 0 \\ 1 & 05:06:00 & 05:06:30 & 9220070 & 4 & 0 \\ 2 & 04:55:00 & 04:45:30 & 9230999 & 1 & 0 \\ 2 & 05:01:00 & 05:01:30 & 9230400 & 2 & 1 \\ 2 & 05:09:00 & 05:09:30 & 9220019 & 3 & 0 \\ 2 & 05:16:00 & 05:16:30 & 9220070 & 4 & 0 \end{bmatrix} \end{array}$$

In the following, we discuss strategies for compression of GTFS relations. Standard compression tools usually perform a byte-wise compression, which means that these tools view a GTFS relation as a stream of bytes. The serialization of GTFS relations in a CSV-file is line-by-line. Therefore, standard tools will compress GTFS relations line-by-line. The first line of M_{EX1}, for instance, is '1,1,04:45:00,04:45:30,9230999,1,0' followed by the second line '2,1,04:51:00,04:51:30,9230400,2,1'. Compressing such lines is inherently inefficient (see our evaluation in Section V): During the traversal of a line, similar items are not kept together, and also cannot easily be identified as similar by standard compression tools, since these tools do not break lines into their items, but rather work on a per-byte level.

An alternative traversal strategy is to compress GTFS relations column-wise, a processing strategy with long traditions inside the database community [29], [30]. Column-oriented database systems have been shown to outperform traditional ones significantly for specific tasks [31], [32]. Moreover, the storage of similar values together allows for very efficient compression, called *vertical compression* in [18]. Avoid changing compression contexts has also been discussed in the Bioinformatics community [33], where FASTQ-files representing information about sequencing of individuals are compressed. In addition to the raw data, FASTQ files contain also metadata, such as quality scores denoting uncertainties occurred during sequence identification. All these streams in the input file are compressed separately, which yields an increased compression ratio and significantly higher compression speeds [33].

Starting with the idea of column-wise compression, we design a compression algorithm for GTFS relations as follows. First, we define a compression technique for a column following a sliding window over the to-be-compressed column. Intuitively, the compressed code for an item is computed based on the previous n items occurring in the column. Second, we generalize this compression technique in such a way that the code for an item in a column can also be computed on previous and current items in other columns. In order to ensure decompressibility, we need to avoid cyclic dependencies for computing codes from other column values.

Definition IV.2 (Self-referential compression queue)**.** *A* Self-referential compression queue *for column i, with $h_i = [x_{i,1}, x_{i,2}, \dots, x_{i,n}]$ of M_{GTFS} with window size w, is a list $[code_1, code_2, \dots, code_n]$, such that $code_j$ is a compressed representation of $x_{i,j}$, computed from the list $[x_{i,j-w}, x_{i,j-(w-1)}, \dots, x_{i,j-1}]$.*

Definition IV.2 ensures two things: 1) The amount of memory is limited by window size w during compression and decompression; 2) unique decompressibility is ensured by referencing previously occurred items only, which can be incrementally decoded at decompression time. We will discuss a simple self-referential compression queue below.

The column *stop_sequence* from M_{EX1} of Example 1 contains a sequence of increasing integers, within a single trip. Therefore, knowing the preceding value of an item in a column can often yield a more compact representation: Instead of encoding the absolute value of a stop_sequence, we encode the difference with the previous item minus 1. That is, we design a self-referential compression queue with window size $w = 1$. Hereby, $code_j = x_{i,j} - x_{i,j-1} - 1$. For the remainder of this example, we assume that $x_{i,0}$ is preset to -1. The first code, $code\_1$, is computed to $0 - (-1) - 1 = 0$. The second code, $code\_2$ is evaluated to $1 - 0 - 1 = 0$. Overall, we obtain the self-referential compression queue $[0, 0, 0, 0, -4, 0, 0, 0]$. Following run-length encoding [34], such a sequence can often be better compressed than the original sequence $[1, 2, 3, 4, 1, 2, 3, 4]$. The decom-

pression of $[0, 0, 0, 0, -4, 0, 0, 0]$ is straight-forward, knowing the initial value $x_{i,0}$. The column $stop\_sequence$ is only one example for how self-referential compression queues can be used for an efficient compression. Another example is the column $arrival\_time$. Encoding, for instance, the temporal difference between 04:51:00 and previous value 04:45:00 (in seconds) consumes less space than the uncompressed storage of 04:51:00. A third example is the prediction of the next stop in column $stop\_id$. Trips along the same route often contain identical subtrips. Thus, given knowledge about the stop-pairs of previous trips, one can often accurately predict the next stop of the current trip (unless the trip goes along a different route). We generalize the concept of self-referential compression queues to referential compression queues, which can refer to other columns for computing codes.

Definition IV.3 (Referential compression queue)**.** *A referential compression queue for column i, with $h_i = [x_{i,1}, x_{i,2}, \ldots, x_{i,n}]$ of M_{GTFS} with window size w, is a list $RCQ_i = [code_1, code_2, \ldots, code_n]$, such that $code_j$ is a compressed representation of $x_{i,j}$, computed from the lists $L_1, \ldots, L_m$, such that*

$$L_k = \begin{cases} [x_{k,j-w}, x_{k,j-(w-1)}, \ldots, x_{k,j-1}, x_{k,j}] & \text{if } k \neq i \\ [x_{k,j-w}, x_{k,j-(w-1)}, \ldots, x_{k,j-1}] & \text{otherwise} \end{cases}$$

It should be noted that in referential compression queues, a code can be computed based on items of other columns *in the same row*. Therefore, in order to ensure decompressibility, we need to avoid cases where a $code_j$ for column i_1 is computed based on $x_{i_2,j}$ and at the same time $code_j$ for column i_2 is computed based on $x_{i_1,j}$. Intuitively, we need to avoid cyclic references. A referentially compressed GTFS relation is defined as follows.

Definition IV.4 (Referentially compressed GTFS relation)**.** *A referentially compressed GTFS relation for M_{GTFS} is a list of referential compression queues $[RCQ_1, \ldots, RCQ_n]$, such that each RCQ_i is a referential compression queue for column i of M_{GTFS}, and there are no cyclic references between the compression queues.*

Our Definition IV.4 provides a framework for compressing GTFS relations. The actual implementation, however, is left open. This mainly concerns the decision about which column is encoded against other columns. We will address this in the next subsection.

B. Implementation

First, we discuss the compression of the three largest relations occurring in GTFS, according to our empirical evaluation in Table II (see Section V). These three relations are stop_times.txt, shapes.txt, and trips.txt. All remaining files are compressed with a simple compression method, since they do not contribute much to the compressed size of a GTFS dataset.

1) Compression of stop_times.txt*:* The file stop_times.txt consists of five or more columns. The five mandatory columns are: trip_id (uniquely identifying a trip for which stop times are encoded), stop_id (the stop whose schedule is described), arrival_time (time when the public transport vehicle is scheduled for arrival), departure_time (time when the public transport vehicle is scheduled for departure), and stop_sequence (an increasing number, identifying the order of stops along a trip). Other (optional) columns contain, for instance, information about wheelchair accessibility and links to shape information for visualization of the trip. In the following, we describe the domain-specific compression of all mandatory columns; the optional columns are compressed with a default compressor.

Algorithm 1 Compression of stop_times.txt

Input: to-be-compressed GTFS relation M_{stop_times} with m columns and n rows, and a windows size w
Output: Referentially compressed GTFS relation RCR_{stop_times}
1: Let $heads = [h_1, h_2 \ldots, h_m]$
2: **for all** $h \in heads$ **do** ▷ Assign a referential compression queue to each head, with the code being a raw string
3: Let $RCQ[h] = [' R' \circ x_{h,1}]$
4: **end for**
5: **for** $2 \leq j \leq n$ **do** ▷ For a head h, below we denote $[x_{h,(j-w)}, \ldots, x_{h,(j-1)}])$ with $W(h)$
6: **for all** $h \in heads$ **do**
7: **if** $h \equiv'$ $trip\_id'$ or $h \equiv'$ $stop\_sequence'$ **then**
8: $append(RCQ[h], encodeINC(x_{h,j}, x_{h,j-1}))$
9: **else if** $h \equiv'$ $stop\_id'$ **then**
10: $append(RCQ[h], encodeSUCC(x_{h,j}, W(stop\_id)))$
11: **else if** $h \equiv'$ $arrival\_time'$ **then**
12: $append(RCQ[h], encodeAT(x_{h,j}, x_{stop\_id,j}, W(arrival\_time), W(departure\_time), W(stop\_id)))$
13: **else if** $h \equiv'$ $departure\_time'$ **then**
14: $append(RCQ[h], encodeDT(x_{h,j}, x_{stop\_id,j}, x_{arrival\_time,j}, W(arrival\_time), W(departure\_time), W(stop\_id)))$
15: **else**
16: $append(RCQ[h],' R' \circ x_{h,j})$
17: **end if**
18: **end for**
19: **end for**
20: **return:** $RCR_{stop\_times} = [RCQ[heads[1]], RCQ[heads[2]], \ldots, RCQ[heads[m]]]$

Algorithm 2 Function $encodeINC$

Input: current item cur and preceding item $prev$
Output: code for cur compressed against $prev$
1: $code =''$
2: Let suf_{cur} be the longest suffix of cur, such that suf_{cur} is an integer number, and $pref_{cur}$ the result of removing suf_{cur} from the end of cur
3: Let suf_{prev} be the longest suffix of $prev$, such that suf_{prev} is an integer number, and $pref_{prev}$ the result of removing suf_{prev} from the end of cur
4: **if** $cur \equiv prev$ **then**
5: $code =' I'$
6: **else if** $pref_{cur} \equiv pref_{prev}$ and $pref_{cur}(0) \neq 0$ and $pref_{prev}(0) \neq 0$ **then**
7: $code =' D' \circ (tonumber(pref_{cur}) - tonumber(pref_{prev}))$
8: **else**
9: $code =' R' \circ cur$
10: **end if**
11: **return:** $code$

The general compression algorithm for stop_times.txt is shown in Algorithm 1. The algorithm receives the GTFS relation M_{stop_times} and window size w as inputs. First, a referential compression queue is initialized for each column in M_{stop_times} (this includes mandatory and optional columns). The queues are stored in the map RCQ, such that $RCQ[h]$ returns the queue for a column with header h. All queues are initialized with the elements from the first line of M_{stop_times}; the symbol 'R' in front of each element encodes a *raw* encoding, which means that the code does not contain references to other elements. Below, we will use additional symbols, to describe how an item is encoded. Storing these symbols is necessary in order to ensure decompressibility, and therefore a lossless compression implementation. After the initialization of the queues, Algorithm 1 proceeds to compress lines 2–n from M_{stop_times}. For each column, we define a particular compression method, which is fine-tuned for the domain values (as stated by the GTFS specification). We describe these column-specific compression steps in detail below.

The columns trip_id and stop_sequence are encoded incrementally with respect to the previously occurred item in the same column, using the function $encodeINC$ shown in Algorithm 2. The idea for $encodeINC$ is to encode only the differences between two consecutive items. We have motivated this referential compression technique in Section IV-A, for the encoding of sequences such as $[1, 2, 3, 4, 5, 6]$, where storing

Algorithm 3 Function $encodeSUCC$

Input: current item cur and preceding items in window $[prev_w, prev_{w-q}, \ldots, prev_1]$
Output: code for cur compressed against $[prev_w, prev_{w-q}, \ldots, prev_1]$
$code = ''$
$pos = -1'$
for $2 \leq j \leq w$ **do**
  if $prev_1 \equiv prev_j$ **then**
    $pos = j$
    break
  end if
end for
if $pos \geq 0$ and $prev_{pos+1} \equiv cur$ **then**
  $code =' M'$
else
  $code =' R' \circ cur$
end if
return: $code$

Algorithm 4 Function $encodeAT$

Input: current arrival time t, current stop_id $stop\_id$, preceding items in arrival_time window $[arr_w, arr_{w-q}, \ldots, arr_1]$, preceding items in departure_time window $[dep_w, dep_{w-q}, \ldots, dep_1]$, and preceding items in stop_id window $[stop_w, stop_{w-q}, \ldots, stop_1]$
Output: code for t compressed against $stop\_id$ and the input windows
$timediff = t - dep_1$ ▷ Difference between current arrival time and previous departure (in s)
for $2 \leq j \leq w$ **do**
  if $stop\_id \equiv stop_{j-1}$ and $stop_1 \equiv stop_j$ and $arr_{j-1} - dep_j \equiv timediff$ **then**
    return: $'M'$
  else if $stop\_id \equiv stop_{j-1}$ and $stop_1 \equiv stop_j$ **then**
    return: $'D' \circ (timediff - (arr_{j-1} - dep_j))$
  end if
end for
return: $'R' \circ cur$

the delta values between the numbers explicitly. However, the situation is slightly complicated for trip_id: Service providers often choose mixed identifiers, consisting of textual part (occurring as a prefix of the identifier) and a number. For instance, trip identifiers for the public transport in Madrid are described as 'FE0010011', 'FE0010012', 'FE0010013', and so on. Thus, our implementation of $encodeINC$ splits the ids into a textual prefix (which can be empty) and a numerical suffix. In the example for Madrid, the common prefix is 'FE', while the numerical suffixes are 0010011, 0010012, and 0010013. The encoding function distinguishes three cases: 1) the prefixes and suffixes are identical, respectively (only the symbol 'I' is stored), 2) the items share a common textual prefix (only the differences between the numerical values are stored), 3) no similarities are identified (the raw item is stored). Finally, the computed code is returned. At decompression time, the codes are sufficient for decoding: If a 'I' is obtained, the item matches the previously decompressed item, if a D is obtained with a delta value, we can simply add the delta to the numerical suffix of the previously decoded item, and raw values are decompressed natively.

Coming back to Algorithm 1, we describe next how the column stop_id is compressed using the function $encodeSUCC$, as shown in Algorithm 3. This function finds the most recent occurrence of the predecessor of cur in the preceding items window and checks, whether the successor is identical to the current item cur. If so, then the item cur is encoded with the symbol 'M'. At decompression time, the item is decoded by looking up the most recent successor of the predecessor. If the item cur does not occur in the window or the predecessors do not match, then cur is encoded as a raw value, using symbol 'R'.

Next, we discuss the compression of arrival_time in Algorithm 1. The function $encodeAT$ is called with the following parameters: Current arrival_time, current stop_id, and the histories of arrival_time, departure_time, and stop_id. The

Algorithm 5 Function $encodeDT$

Input: current departure time t, current stop_id $stop\_id$, current arrival_time $arrival\_time$, preceding items in arrival_time window $[arr_w, arr_{w-q}, \ldots, arr_1]$, preceding items in departure_time window $[dep_w, dep_{w-q}, \ldots, dep_1]$, and preceding items in stop_id window $[stop_w, stop_{w-q}, \ldots, stop_1]$
Output: code for t compressed against $stop\_id$, $arrival\_time$, and the input windows
$timediff = t - arrival\_time$ ▷ Difference between current departure time and current arrival time (in s)
if $timediff \equiv 0$ **then**
  return: $'A'$
end if
for $2 \leq j \leq w$ **do**
  if $stop\_id \equiv stop_j$ $dep_j - arr_j \equiv timediff$ **then**
    return: $'M'$
  else if $stop\_id \equiv stop_j$ **then**
    return: $'D' \circ (timediff - (dep_j - arr_j))$
  end if
end for
return: $'S' \circ timediff$

Algorithm 6 Compression of shapes.txt

Input: to-be-compressed GTFS relation M_{shapes} with m columns and n rows, and a windows size w
Output: Referentially compressed GTFS relation RCR_{shapes}
Let $heads = [h_1, h_2 ..., h_m]$
for all $h \in heads$ **do** ▷ Assign a referential compression queue to each head, with the code being a raw string
  Let $RCQ[h] = [' R' \circ x_{h,1}]$
end for
for $2 \leq j \leq n$ **do** ▷ For a head h, below we denote $[x_{h,(j-w)}, \ldots, x_{h,(j-1)}])$ with $W(h)$
  for all $h \in heads$ **do**
    if $h \equiv'$ $shape\_id'$ or $h \equiv'$ $shape\_pt\_sequence'$ **then**
      $append(RCQ[h], encodeINC(x_{h,j}, x_{h,j-1}))$
    else if $h \equiv'$ $shape\_dist\_traveled'$ **then**
      $append(RCQ[h], encodeDIST(x_{h,j}, x_{shape\_pt\_lat,j}, x_{shape\_pt\_lon,j}, x_{shape\_pt\_lat,j-1}, x_{shape\_pt\_lon,j-1}, x_{h,j-1})$
    else
      $append(RCQ[h],' R' \circ x_{h,j})$
    end if
  end for
end for
return: $RCR_{shapes} = [RCQ[heads[1]], RCQ[heads[2]], \ldots, RCQ[heads[m]]]$

compression in function $encodeAT$, as shown in Algorithm 4, proceeds as follows. The temporal difference (in seconds) between the previous departure_time at $stop\_1$ and current arrival_time at $stop\_id$ is computed. The historical window is traversed to find a recent occurrence of a subtrip from $stop\_1$ to $stop\_id$. If the travel time (the difference between departure_time and arrival_time) is identical, then a perfect match is encoded with 'M'. Otherwise, the time difference between both travel times is encoded as a delta value, using symbol 'D'. If no previous subtrip between $stop\_1$ and $stop\_id$ is found, then the arrival_time is encoded as a raw value, using symbol 'R'.

The column departure_time is encoded referentially as shown in Algorithm 5. The function $encodeDT$ is called with the following parameters: Current departure_time, current stop_id, current arrival_time, and the histories of arrival_time, departure_time, and stop_id. The algorithm finds the most recent historical description of a stop at stop_id. If the idle time at the stop (difference between departure_time and arrival_time) is identical, then the time is encoded as a perfect match 'M'. Otherwise, if the time does not match, the idle time is encoded referentially. If there is no previous description of a stop at stop_id in the window, the departure time is encoded referentially towards the current arrival_time.

All remaining columns are encoded with raw values initially. In the end, all referential compression queues are compressed with a standard entropy encoder.

2) Compression of shapes.txt*:* The file shapes.txt consists of four or more columns. The four mandatory columns are:

Algorithm 7 Function $encodeDIST$

Input: current item cur, latitudes lat_1,lat_2, longitudes lon_1,lon_2, and previous distance $dist$
Output: code for cur compressed against $prev$

Let $a = sin^2(\frac{lat_2 - lat_1}{2}) + cos(lat_1) * cos(lat_2) * sin^2(\frac{lon_2 - lon_1}{2})$
Let $c = 2 * atan2(\sqrt{a}, \sqrt{1-a})$
Let $stepKM = c * 6,371$
Let $stepMI = c * 3,959$
if $dist + stepKM \equiv cur$ **then**
  return: $'K'$
else if $dist + stepMI \equiv cur$ **then**
  return: $'M'$
else if $dist + \frac{stepKM}{1000} \equiv cur$ **then**
  return: $'E'$
else if $dist + \frac{stepMI}{5280} \equiv cur$ **then**
  return: $'F'$
else
  return: $'R' \circ cur$
end if

Algorithm 8 Compression of trips.txt

Input: to-be-compressed GTFS relation M_{trips} with m columns and n rows
Output: Referentially compressed GTFS relation RCR_{trips}

Let $heads = [h_1, h_2..., h_m]$
for all $h \in heads$ **do** ▷ Assign a referential compression queue to each head, with the code being a raw string
  Let $RCQ[h] = ['R' \circ x_{h,1}]$
end for
for $2 \leq j \leq n$ **do**
  for all $h \in heads$ **do**
    if $h \in \{'route\_id','service\_id','trip\_id','shape\_id','block\_id'\}$ **then**
      $append(RCQ[h], encodeINC(x_{h,j}, x_{h,j-1}))$
    else
      $append(RCQ[h],' R' \circ x_{h,j})$
    end if
  end for
end for
return: $RCR_{trips} = [RCQ[heads[1]], RCQ[heads[2]], \ldots, RCQ[heads[m]]]$

shape_id (uniquely identifying a shape which is modeled), shape_pt_lat (describes the latitude for a single point of the shape), shape_pt_lon (describes the longitude for a single point of the shape), and shape_pt_sequence (an increasing number, identifying the order of points along a shape). A fifth column, shape_dist_traveled, is optional and contains information about the distance between two consecutive points of a shape. In the following, we describe the domain-specific compression of three columns in shapes.txt; all other columns are compressed with a default compressor.

The general compression algorithm for shapes.txt is shown in Algorithm 6. The overall structure is similar to that of Algorithm 1 for compressing stop_times.txt. Therefore, we only describe the differences here. The two columns shape_id and shape_pt_sequence are encoded incrementally with respect to previously occurred item in the same column, reusing the function $encodeINC$ shown in Algorithm 2. The column shape_dist_traveled is encoded with the function $encodedDIST$ and is explained below; all remaining columns are encoded with raw values.

The encoding of column shape_dist_traveled deserves explanation. This value shape_dist_traveled represents the real distance traveled along the route so far, from the first shape point. The major problem with the definition of this column (in the GTFS specification) is twofold: First, it is completely left open, which units are used to model the distance (the GTFS reference suggests feet and kilometers as examples). Second, it is not stated, whether real distance refers to the distance traveled by the public transportation vehicle or to the line-by-sight distance. However, since the distance is (from a database point of view) often redundant, given information about previously occurred latitude/longitude points, we try to estimate the distance (and the unit) of this column and encode the deviation as a delta, whenever possible, using the Haversine formula [35]. The implementation is defined in Algorithm 7. We probe four different units (kilometer, miles, meters, and feet): If any of them represents the actual increase in distance, we encode this information with a single symbol, since the distance can be easily recomputed from the previous distance and latitude/longitude values. Otherwise, if none of the units fits, we encode the shape_dist_traveled as a raw value.

All remaining columns are encoded with raw values initially. In the end, all referential compression queues are compressed with a standard entropy encoder.

3) Compression of trips.txt*:* The file trips.txt consists of three or more columns. The three mandatory columns are: route_id (uniquely identifying a modeled route), service_id (describes the set of dates when a service is available), and trip_id (a unique identifier of the trip). There are several optional columns, for instance, trip_headsign (encoding the text that represents the trip's destination), shape_id (associating the trip to shape describing the trip), and other columns for identifying wheelchair accessibility and bike allowance. In the following, we describe the domain-specific compression of five columns. Additional columns, which may be added by service providers, are compressed with a default compressor.

The general compression algorithm for trips.txt is shown in Algorithm 8. The overall structure is similar to that of Algorithm 1 for compressing stop_times.txt. Thus, only the differences are described. In general, file trips.txt mainly consists of identifiers, These identifiers are efficiently compressed by our efficient identifier compression technique from Algorithm 2, given they are sorted (which is often the case). The remaining columns are stored using raw values. Most of these remaining columns have very short values (often only one byte), where referential compression does not pay off. In the end, all referential compression queues are compressed with a standard entropy encoder.

4) Compression of other GTFS relations: All other files are compressed using a standard compression algorithm. These files are usually orders of magnitude smaller than the three previously discussed files and therefore do not dominate the overall size of the compressed GTFS dataset. Moreover, many of the remaining files are less structured, which makes it difficult to design an efficient compression algorithm.

C. Overall comments on the implementation

First, our technique requires an implementation of the data structure *referential compression queue*. Since this is essentially a sliding window, the implementation is rather easy. The critical part is an efficient look-up of previously seen elements. In our implementation, we have used a double-ended queue (deque in c++). Recently seen elements are stored in the queue and the length of the queue is restricted to the sliding window size. In general, a hash-based implementation might further speed up the process of finding previously seen elements, by avoiding repetitive traversals of the queue.

The implementation of our encoding functions is straightforward, given their algorithmic descriptions and does not pose any particular challenges. It should be noted that we used p7zip (http://www.7-zip.de/download.html) as a standard entropy encoded for the compression of pre-compressed columns. The compression tool was used with its default parameters.

GTFSCompress fits the life cycle of a GTFS dataset as follows: 1) The dataset for a region/temporal period is created.

Dataset	Uncompressed size (MB)	zip-compressed size (MB)	Number of routes	Number of trips	Number of stops	Number of shapes
BERLIN	195	27	**1,452**	220,275	13,159	**0**
BOSTON	272	35	217	176,169	8,399	1,055
CHICAGO	398	50	134	91,170	11,707	1,696
MADRID	100	14	207	76,429	4,655	411
MILANO	345	27	155	34,988	4,866	754
MONTREAL	276	47	222	148,738	9,545	1,022
PARIS	**702**	**98**	1,083	**568,655**	**26,965**	**0**
PORTLAND	168	24	91	33,147	6,919	1,237
SANFRANCISCO	**51**	**8**	**84**	**28,086**	**3,596**	498
WASHINGTON	116	19	298	50,420	10,840	**2,235**

TABLE I: Overview on the ten GTFS datasets used in this study. Minimum and maximum values are highlighted in bold. PARIS is the largest dataset (at 702 MB), with the highest number of trips and stops. SANFRANCISCO is the smallest dataset in our study (at 51 MB).

Dataset	BERLIN	BOSTON	CHICAGO	MADRID	MILANO	MONTREAL	PARIS	PORTLAND	SANFRANCISCO	WASHINGTON
agency.txt	0.006	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
calendar.txt	0.079	0.017	0.005	0.000	0.000	-	0.284	-	0.000	-
calendar_dates.txt	2.576	0.045	0.003	0.000	0.028	0.003	1.564	0.009	0.000	0.008
frequencies.txt	-	0.001	0.000	2.569	-	0.339	-	-	-	-
route_xref.txt	-	-	-	-	-	-	-	-	-	3.227
routes.txt	0.061	0.013	0.013	0.026	0.014	0.016	0.101	0.006	0.003	0.008
shapes.txt	**-**	**16.255**	**31.963**	**1.919**	**7.208**	**1.871**	**-**	**34.062**	**2.090**	**24.919**
stop_features.txt	-	-	-	-	-	-	-	1.144	-	-
stop_times.txt	**179.722**	**237.147**	**358.442**	**90.570**	**313.874**	**266.413**	**672.345**	**129.469**	**47.575**	**85.134**
stops.txt	0.817	0.635	1.330	0.674	0.533	1.060	2.771	1.156	0.212	0.734
transfers.txt	0.364	0.001	0.003	-	0.000	-	1.705	0.051	-	-
trips.txt	**11.168**	**17.801**	**5.701**	**3.038**	**22.963**	**5.745**	**23.258**	**1.094**	**1.307**	**1.983**
Other files in dataset	0.000	0.000	0.019	0.000	0.000	0.005	0.000	0.006	0.009	0.043

TABLE II: Uncompressed file sizes in MByte for each file in the ten GTFS datasets. Files that are not included in a dataset are marked with '-'. The three largest files (average size) are highlighted in bold. The largest file is stop_times.txt (often comprising 90% of a GTFS dataset), followed by trips.txt and shapes.txt.

2) GTFSCompress is used to compress the file initially on the hard disk. 3) Whenever the file needs to be accessed, GTFSCompress is used to decompress the dataset into main memory and make it accessible for other applications. 4) Once the application does not need access to the dataset anymore, the uncompressed dataset is removed from main memory, while the compressed dataset remains on hard disk.

V. Evaluation

We perform an evaluation on ten real-world GTFS datasets for different regions. The datasets are described in Section V-A. We evaluate our implementation GTFSCompress against standard compression tools in Section V-B.

A. Datasets

For the evaluation we selected ten datasets from the GTFS feeds at GTFS-Data-Exchange[4]. Our datasets cover small cities to larger metropolitan areas: BERLIN, BOSTON, CHICAGO, MADRID, MILANO, MONTREAL, PARIS, PORTLAND, SANFRANCISCO, and WASHINGTON. We show basic statistics about the datasets in Table I. The largest (uncompressed) dataset is PARIS, which needs more than 700 MB of storage. The smallest dataset in our collection is SANFRANCISCO (at around 50 MB).

[4]http://www.gtfs-data-exchange.com

In Table II, we compare the size of components from the GTFS datasets for the ten regions. Overall, it can be concluded that the file size for a single component (see Table I for the total size of the datasets) can be tremendously high. The feed for PARIS, for instance, uses approx. 700 MB of storage; yet more than 90% of the storage is induced by the file stop_times, which encodes schedules for stops at trip level. Similarly, the required storage for most datasets is dominated by the storage for stop_times.txt. The second largest components are trips.txt and shapes.txt. These three components together make up for about 98% of all data in the evaluation datasets. Therefore, in Section IV we have focused on a customized compression of these three components.

B. GTFSCompress

The efficiency of our referential compression queues depends on how much memory is used for the referential compression process: A larger window usually implies a higher compression ratio, but also higher compression times and more memory usage during compression and decompression. In Fig. 1 (left), we show the compression ratio for all ten datasets with a varying window size. The compression ratio clearly increases with window size. For window sizes smaller than 50–100, the compression ratio becomes significantly smaller. Window sizes above 1000 often do not increase the compression ratio anymore. In Fig. 1 (right), we present the compression

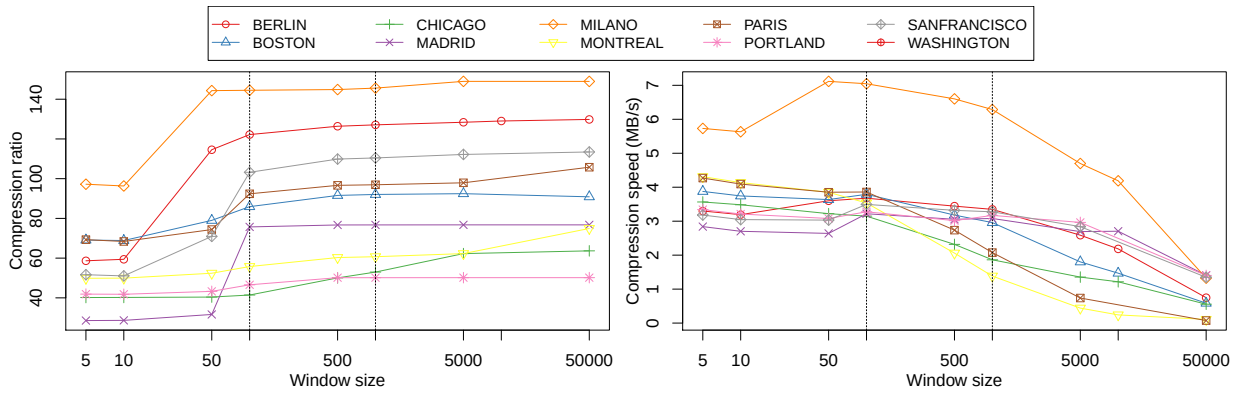

Fig. 1: Compression ratio (left) and compression speed (right) with varying window size (5–50000). An increasing window size allows for higher compression, but comes at the price of lower compression speeds. Increasing the window size beyond 1000 lines does not increase compression ratios significantly.

	Compression ratio					Compression speed				
Dataset	**zip**	**bzip2**	**LZMA2**	**GTFSCompress100**	**GTFSCompress1000**	**zip**	**bzip2**	**LZMA2**	**GTFSCompress100**	**GTFSCompress1000**
BERLIN	8.77	12.77	21.97	122.20	**127.10**	4.44	**5.37**	1.77	3.68	3.35
BOSTON	9.24	13.54	25.46	85.92	**92.06**	**5.40**	5.34	2.24	3.86	3.01
CHICAGO	9.09	9.66	21.33	41.43	**52.98**	**7.17**	5.37	3.09	3.20	1.89
MADRID	7.22	7.26	14.60	75.71	**76.72**	4.58	**5.32**	1.45	3.29	3.15
MILANO	13.83	17.77	31.33	144.50	**145.59**	**10.38**	6.01	4.20	7.12	6.35
MONTREAL	7.38	9.49	14.56	55.77	**60.71**	5.22	**5.51**	1.60	3.61	1.41
PARIS	8.45	11.45	17.02	92.39	**96.94**	5.32	**5.53**	1.85	3.86	2.07
PORTLAND	7.93	9.29	21.76	46.57	**50.15**	**6.61**	5.62	2.64	3.32	3.21
SANFRANCISCO	7.29	8.40	24.14	103.21	**110.44**	4.81	**5.59**	1.82	3.57	3.35
WASHINGTON	7.00	8.24	19.97	48.28	**48.65**	4.93	**5.43**	1.51	3.12	2.80
Average	8.62	10.79	21.21	81.60	**86.13**	**5.88**	5.51	2.22	3.86	3.06

TABLE III: Comparison of compression ratio between GTFSCompress and standard compression tools. The highest compression ratio and the highest compression speed are marked in bold. GTSCompress1000 always receives the highest compression ratios, followed by GTFSCompress100. The standard compressor zip allows for the fastest compression. GTFSCompress instances are usually around 50–70% slower than zip.

speed for all ten datasets with a varying window size. With an increasing window size, the compression speed gradually reduces, since more and more historical items have to be checked during the compression process. The compression speed is stable for window sizes smaller than 100–1000. In the following, we test two instantiations of GTFSCompress: One with window size 100 and another one with window size 1000. Still, in our implementation, the window size can be chosen by the user.

We compare GTFSCompress against three compression tools as implemented in 7zip[5]:

- **zip**: Based on LZ77 algorithm (7za a -tzip).
- **bzip2**: Based on Burrows-Wheeler Transformation (7za a -tbzip2).
- **LZMA2**: Improved and optimized version of LZ77 algorithm (7za a -t7z -m0=LZMA2). In LZMA2, compression is improved over LZ77 by using a longer history buffer (up to 4 GB), optimal parsing, shorter codes for recently repeated matches, literal exclusion after matches, and arithmetic coding [36].

The main results of our experiments are reported in Table III. GTFSCompress, tested with window size 100 and 1000, clearly outperforms the standard compression tools regarding compression ratio. Compared to zip, the standard format for compressing GTFS datasets, the compression ratio of GTFSCompress is approx. one order of magnitude higher (86.1 compared to 8.62). The difference with other standard compression tools is still in the order of 4–8 times better compression. Remarkably, GTFSCompress1000 consistently obtains the highest compression ratios for all datasets in our study.

Regarding the compression speed, GTFSCompress is slower than other standard compression tools. The fastest compressor is zip, followed by bzip2. However, one should keep in mind that the process of compressing a dataset is a one-time event: Usually, once a dataset is compressed, it is transferred through a network to clients and stored there for further use. Therefore, we think that the advantage of GTFSCompress in compression ratio outweighs the slower compression speed. In addition, we would like to emphasize that our implementation, which is available for free academic use, was programmed in a clear way (to correctly present implementations of algorithms proposed in this paper), without tricks for optimization. We think that the compression speed of GTFSCompress can be further improved by using standard data structures, e.g. bloom filter, to check whether searching a previously occurring item is useful or not. Furthermore, a considerable amount of time is spent on parsing the comma-separated files; a process which could be implemented more efficiently.

In Figure 2, we report decompression experiments for our

[5]http://www.7-zip.org/

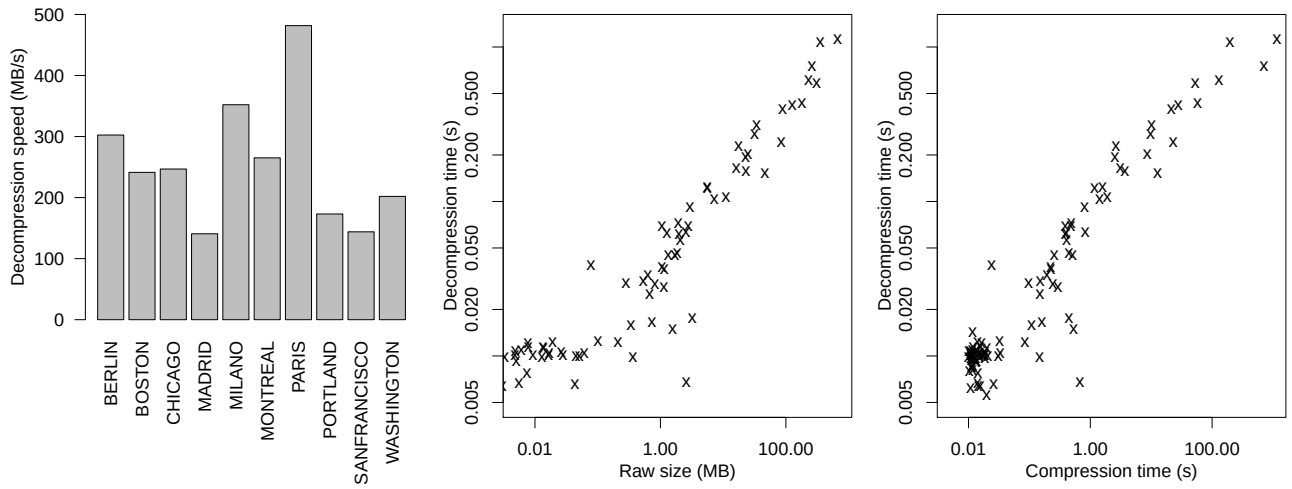

Fig. 2: Decompression of GTFS datasets with GTFSCompress1000. Left: Decompression speed for datasets. Middle: Decompression time increases with raw size of the file in the dataset. Right: Decompression time correlates with compression time, but is up to 100 times faster.

datasets. In Figure 2(left), the decompression speed is between 200 and 500 MB/s. In our experiments, the largest dataset, PARIS, can be decompressed within 1.5 seconds using our compression algorithm, all other datasets can be accessed within a single second. In Figure 2(middle), we plot the decompression speed against the raw size of files inside the datasets. It can be seen that the decompression time largely depends on the raw size of the input, independent of the file type. Similarly, as shown in Figure 2(right), the decompression time strongly correlates with compression time, only that the decompression is faster by approx. two orders of magnitude. This is a valuable property for read-only access to GTFS datasets, where data is compressed once but decompressed frequently.

VI. Conclusions

We developed a new technique for lossless compression of public transportation schedules encoded in GTFS feeds. Standard compression tools only achieve very low compression ratios, between 8–20, depending on the chosen compression scheme. The major reason for poor compression performance is that these tools cannot efficiently identify and exploit redundancies in the data. In this work we have introduced the algorithms for GTFSCompress, which is based on the idea of compressing columns in GTFS relations referentially against other columns, leading to a data structure called referential compression queues. Our experiments with ten real-world GTS datasets show that transportation schedules can actually be compressed up to a factor of 145 (86.13 in average). We think that our work on GTFSCompress makes it feasible to store large amounts of transit schedules on commodity hardware. The applicability of GTFSCompress is summarized as follows:

1) The compression speed is lower than that of the best standard compressors. However, since compression is often a one-time event, we believe that this is not a severe limitation in practice.
2) When changing the format significantly, e.g. changing the data model from relational to XML, our method needs to be adapted; while standard compression tools can still work out of the box. As long as the data model remains the same, i.e. with one relation per file, our compression method does not need adaptations.
3) The overhead for adapting our algorithms to different datasets is zero, as long as the underlying data representation is a relational data model. If previously unknown columns are added to feeds, they are compressed by a default column-wise compressor. Moreover, all column-compression algorithms developed in this paper fall back to encode raw values, in case the current value cannot be encoded efficiently (or is unexpected). Since we post-process the output with an entropy encoder, the results are still well compressed. This makes our approach much more applicable than the techniques in original reference [34], which needs many adaptions and user interventions in case of changing the GTFS files. Compared to that method, our method simply works out of the box.

We discuss several directions for future work. First, the sort order of tuples in GTFS relations can have a significant impact on the compression ratio. In this study, we did not change the order of tuples, in order to achieve true lossless compression. Additional experiments could be performed by compressing different orderings. Second, our analysis revealed that one major limitation of GTFSCompress is the compression of latitude/longitude pairs. It should be possible to incorporate specific compression tools for geo-coordinates [37], [38] to further increase compression ratios. Third, using the techniques presented in this paper, it might become possible to manage all historical public transportation schedules for many regions efficiently in a main memory database for information retrieval. This can lead towards new insights and lays an important foundation for handling big transportation data.

References

[1] T. Litman, "Evaluating public transit benefits and costs," *Victoria Transport Policy Institute*, vol. 65, 2011.
[2] M. L. Anderson, "Subways, strikes, and slowdowns: The impacts of public transit on traffic congestion," National Bureau of Economic Research, Tech. Rep., 2013.
[3] D. He, H. Liu, K. He, F. Meng, Y. Jiang, M. Wang, J. Zhou, P. Calthorpe, J. Guo, Z. Yao *et al.*, "Energy use of, and co 2 emissions from chinas urban passenger transportation sector–carbon mitigation scenarios upon the transportation mode choices," *Transportation Research Part A: Policy and Practice*, vol. 53, pp. 53–67, 2013.

[4] T. Litman, "Comprehensive evaluation of energy conservation and emission reduction policies," *Transportation Research Part A: Policy and Practice*, vol. 47, pp. 153–166, 2013.

[5] B. E. Saelens, A. Moudon, B. Kang, P. Hurvitz, and C. Zhou, "Higher physical activity is directly related to public transit use," *Am J Public Health*, 2013.

[6] S. Gisdakis, V. Manolopoulos, S. Tao, A. Rusu, and P. Papadimitratos, "Secure and privacy-preserving smartphone-based traffic information systems," *Intelligent Transportation Systems, IEEE Transactions on*, vol. 16, no. 3, pp. 1428–1438, June 2015.

[7] J. Wong, "Leveraging the general transit feed specification for efficient transit analysis," *Transportation Research Record: Journal of the Transportation Research Board*, no. 2338, pp. 11–19, 2013.

[8] Google, "General Transit Feed Specification," 2013. [Online]. Available: https://developers.google.com/transit/gtfs/

[9] A. Thiagarajan, J. Biagioni, T. Gerlich, and J. Eriksson, "Cooperative transit tracking using smart-phones," in *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2010, pp. 85–98.

[10] S. J. Barbeau, P. L. Winters, N. L. Georggi, M. A. Labrador, and R. Perez, "Travel assistance device: utilising global positioning system-enabled mobile phones to aid transit riders with special needs," *Intelligent Transport Systems, IET*, vol. 4, no. 1, pp. 12–23, 2010.

[11] J. Zhao, M. Frumin, N. Wilson, and Z. Zhao, "Unified estimator for excess journey time under heterogeneous passenger incidence behavior using smartcard data," *Transportation Research Part C: Emerging Technologies*, vol. 34, no. 0, pp. 70 – 88, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0968090X13001101

[12] A. Carrel, P. S. Lau, R. G. Mishalani, R. Sengupta, and J. L. Walker, "Quantifying transit travel experiences from the users perspective with high-resolution smartphone and vehicle location data: Methodologies, validation, and example analyses," *Transportation Research Part C: Emerging Technologies*, no. 0, pp. –, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0968090X15001060

[13] A. Owen and D. M. Levinson, "Modeling the commute mode share of transit using continuous accessibility to jobs," *Transportation Research Part A: Policy and Practice*, vol. 74, no. 0, pp. 110 – 122, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0965856415000191

[14] B. Ferris, K. Watkins, and A. Borning, "Onebusaway: results from providing real-time arrival information for public transit," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 1807–1816.

[15] H. Bast, P. Brosi, and S. Storandt, "TRAVIC: a visualization client for public transit data," in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Dallas/Fort Worth, TX, USA, November 4-7, 2014*, 2014, pp. 561–564. [Online]. Available: http://doi.acm.org/10.1145/2666310.2666369

[16] T. Szincsak and A. Vagner, "Data Structure to Store GTFS Data Efficiently on Mobile Devices," *JOURNAL OF COMPUTER SCIENCE AND SOFTWARE APPLICATION*, vol. 1, no. 1, 2014.

[17] J. Makler, M. Harvey, S. Callas, K. Tufte, and R. Peterson, "Arriving next on track 1: Online archive for geospatial transit performance data," *Transportation Research Record: Journal of the Transportation Research Board*, no. 2442, pp. 37–43, 2014.

[18] H. Plattner, "A common database approach for OLTP and OLAP using an in-memory column database," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, 2009, pp. 1–2. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559846

[19] K. A. Tufte, R. L. Bertini, and M. Harvey, "Evolution and usage of the portal data archive: A ten-year 2 retrospective 3," in *Transportation Research Board 94th Annual Meeting*, no. 15-4761, 2015.

[20] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. IT-23, no. 3, pp. 337–343, May 1977.

[21] A. Moffat, "Implementing the PPM data compression scheme," *IEEE Transactions on Communications*, vol. COM-38, no. 11, pp. 1917–1921, Nov. 1990.

[22] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," 1994.

[23] S. Wandelt, U. Leser *et al.*, "Adaptive efficient compression of genomes." *Algorithms for Molecular Biology*, vol. 7, no. 30, pp. 1–9, 2012.

[24] D. Le Gall, "Mpeg: A video compression standard for multimedia applications," *Communications of the ACM*, vol. 34, no. 4, pp. 46–58, 1991.

[25] J. Muckell, J.-H. Hwang, V. Patil, C. T. Lawson, F. Ping, and S. S. Ravi, "SQUISH: An Online Approach for GPS Trajectory Compression," in *COM.Geo*. New York, NY, USA: ACM, 2011, pp. 13:1–13:8. [Online]. Available: http://doi.acm.org/10.1145/1999320.1999333

[26] S. Wandelt and X. Sun, "Efficient compression of 4d-trajectory data in air traffic management," *Intelligent Transportation Systems, IEEE Transactions on*, vol. 16, no. 2, pp. 844–853, April 2015.

[27] V. V. Makkapati and P. R. Mahapatra, "Extreme compression of weather radar data," *Geoscience and Remote Sensing, IEEE Transactions on*, vol. 45, no. 11, pp. 3773–3783, 2007.

[28] A. Deligiannakis, Y. Kotidis, and N. Roussopoulos, "Compressing historical information in sensor networks," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004, pp. 527–538.

[29] D. S. Batory, "On Searching Transposed Files," *ACM Trans. Database Syst.*, vol. 4, no. 4, pp. 531–544, Dec. 1979. [Online]. Available: http://doi.acm.org/10.1145/320107.320125

[30] S. Khoshafian, G. P. Copeland, T. Jagodis, H. Boral, and P. Valduriez, "A Query Processing Strategy for the Decomposed Storage Model," in *Proceedings of the Third International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1987, pp. 636–643. [Online]. Available: http://dl.acm.org/citation.cfm?id=645472.655555

[31] P. A. Boncz and M. L. Kersten, "MIL Primitives for Querying a Fragmented World," *The VLDB Journal*, vol. 8, no. 2, pp. 101–119, Oct. 1999. [Online]. Available: http://dx.doi.org/10.1007/s007780050076

[32] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column-oriented database systems," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1664–1665, Aug. 2009. [Online]. Available: http://dx.doi.org/10.14778/1687553.1687625

[33] S. Deorowicz and S. Grabowski, "Compression of DNA sequence reads in FASTQ format," *Bioinformatics*, vol. 27, no. 6, pp. 860–862, 2011. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/27/6/860.abstract

[34] S. Golomb, "Run-length encodings (corresp.)," *Information Theory, IEEE Transactions on*, vol. 12, no. 3, pp. 399–401, Jul 1966.

[35] R. W. Sinnott, "Virtues of the Haversine," *Sky and Telescope*, vol. 68, no. 2, pp. 159+, 1984.

[36] M. Mahoney, *Data Compression Explained*, 2014. [Online]. Available: http://mattmahoney.net/dc/dce.html

[37] P. Cudre-Mauroux, E. Wu, and S. Madden, "Trajstore: An adaptive storage system for very large trajectory data sets." in *ICDE*. IEEE, 2010, pp. 109–120. [Online]. Available: http://dblp.uni-trier.de/db/conf/icde/icde2010.html#Cudre-MaurouxWM10

[38] M. Chen, M. Xu, and P. Franti, "Compression of gps trajectories using optimized approximation," in *Pattern Recognition (ICPR), 2012 21st International Conference on*, Nov 2012, pp. 3180–3183.

Sebastian Wandelt works as a professor at the School of Electronic and Information Engineering at Beihang University. He received a Ph.D. degree in computer science from Hamburg University of Technology in 2011. His research interests are scalable data management, compressing/searching large collections of objects, and Semantic Web reasoning techniques.

Xiaoqian Sun is an associate professor with the School of Electronic and Information Engineering at Beihang University. She obtained her Ph.D. in Aerospace Engineering from Hamburg University of Technology in 2012. Her research interests mainly include air transportation networks, multi-modal transportation, and multi-criteria decision analysis.

Yanbo Zhu is a senior scientist at Beihang University and the vice-president of the Aviation Data Communication Corporation. He obtained his Ph.D. degree in Electronics Engineering from Beihang University in 2009. His main research interests are CNS (Communication, Navigation, Surveillance), ATM (Air Traffic Management), airlines operation services and support.