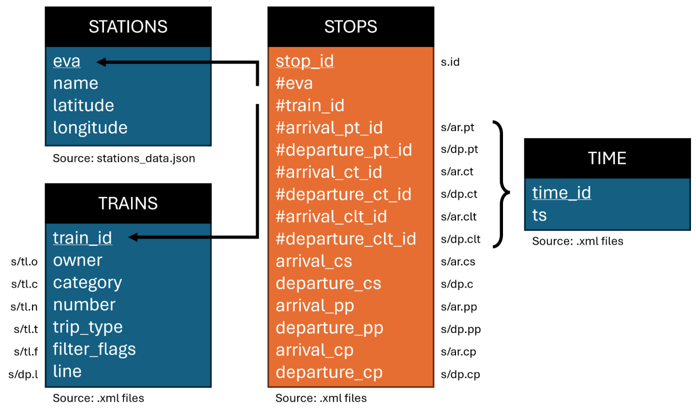
Report: Exercise on Berlin Public Transport Data Analysis

Leonard Püschel (0519780), Darko Cutkovic (0519753), Rebecca Merdes (0456749)

Task 1

Task 1.1

Here is our star schema:



It contains a fact table for the stops and three dimension tables, one for the stations, one for the trains and one for the time. The symbol "#" denotes a foreign key. Primary keys are underlined. The data source for each table is denoted below the attributes. For attributes where it is not obvious, the xml-path is outlined next to the attribute. Tables have this many entries:

STATIONS: 133, TRAINS: 84.083, TIME: 60.384, STOPS: 2.085.038

Task 1.2

Schema Matching Concepts

The main difficulty regarding this point was to put together the data from the different sources. The data of the stations were given in "stations_data.json" and the rest of the data was given in the "timetable" and "timetable_changes" files. Furthermore, the main identifier of station (EVA number) was only given in the "timetable_changes" and in the "stations_data.json", but not in the "timetables". Here is our conceptual approach to overcome this:

stations -> timetables -> timetables_changes
In this approach, the first step is to create the stations table from the json file. After that, create the stops table. Insert the pt and pp values from the planned timetables. To match the stop to a station, a string matcher on the station has to be used (explained below). Although these tables are matched with the station name string, we thought that the EVA number is a better identifier for the stations and thus it is used as PK/FK in the respective tables. After the insertion of the planned values, insert the change values from the timetable changes and use the stop_id to insert at the correct rows. We assumed that stop_id values that exist in timetable_changes, but not in timetables are errors and thus, these ids are ignored. We also considered another approach:

stations -> timetable_changes -> timetables
In this approach, the first step would also be to create the stations table from the json file. After that, create the stops table by iterating the timetable_changes, only insert the ct, clt, cs and cp values and match the station with the EVA number. Finally, iterate the timetable files and insert the missing pt and

pp values by using the stop_id value to find the correct row. Although this approach doesn't require string matching it fails because it does not capture trains that arrive or depart on time as they have no entries in timetable_changes.

String Matcher
Our string matcher is partly hardcoded when it comes to domain-specific aspects. We argue that the scalability of our pipeline should only consider the 'train' domain and thus we justify the hardcoded parts. To compare station names with each other, the strings are normalized first. The normalization process …

- standardizes German Umlaute to double vowel notation
- standardizes naming conventions regarding words like 'straße', 'strasse', 'bf', 'bahnhof', etc…
- and then removes stop words like 'berlin', 's' and 'u'

For a details, see the implementation of the function 'normalize_name' in 'station_matching.py'. After normalization is done, the resulting string are compared which each other. Our comparison metric is:

$$sim \ = \ max\left(\frac{2M}{|A| + |B|}, \frac{|tokens(A) \cap tokens(B)|}{max(|tokens(A)|, |\ tokens(B)|}\right)$$

where M denotes the maximum amount of consecutive letters present in both strings. This is also robust to interruptions in between a word. For example, comparing "AlexaNderplatz" AlexaMderplatz" would yield M=13. The tokens function splits each string into a set of substrings that were separated by whitespaces in the source string. Thus, the similarity measure is invariant to tokens ordered differently in two strings, e. g. comparing "Bahnhof Alexanderplatz" and "Alexanderplatz Bahnhof" would yield a similarity of 100%. But due to the first term, the similarity measure is also somewhat robust to typos. An alternative approach would be to use trigramms as explained in the lecture.

To use this measure in the context of the pipeline, we set some threshold between 0 and 1. To match station names we compare the station name in the xml files with all station names in the json file and create a match according to the maximum number of similarity but only if this similarity is greater than the configured threshold. We chose a threshold of 0.75. As stated in the ISIS forum, we adjusted the name of two stations in the "stations_data.json". (forum post)

Entity Linking and Deduplication
We have found duplicates of the stop_id attribute in the timetables as well as the timetable_changes files. Thus, there was a need for a deduplication strategy.

timetables
We've decided to implement a majority vote in case of multiple stop_id values for these files. The vote happens for each attribute independently. We argue that all values are as important as other ones regardless whether they are from earlier or later ones. Thus, we think that this is the most effective way to make the ETL pipeline robust for errors.

timetable_changes
As we are assuming that values from later snapshots are more accurate to what happened in the real world (see assumptions above), we implemented a "last non-null write" deduplication strategy that overwrites existing values if the incoming values are stemming from an xml file which is from a later snapshot and if incoming values are not null. We argue that this captures the importance of newer values as well as being robust to errors (e. g. newer values being null).

Task 2

Conceptually, the task builds on Task 1. However, our notebook contains some extra queries that we liked to use to investigate the data even further. Markdown cells are additionally used to declare assumptions made in that query. You will find some notes beyond the Notebooks contents here:

Task 2.1

This query retrieves a station's unique EVA identifier and its geographic coordinates based on the provided station name. We made the assumption that user input might be inconsistent in terms of letter casing. Therefore, we designed the query to be case-insensitive. Additionally, we assumed that while the query is initiated by a name, returning the name itself in the result set was redundant.

Task 2.2

During implementation, we considered various geographic distance metrics, such as the Haversine formula, which accounts for the Earth's curvature. However, we decided that the Euclidian metric is sufficient for the relatively small metropolitan area of Berlin. It returns the station with the minimum calculated value.

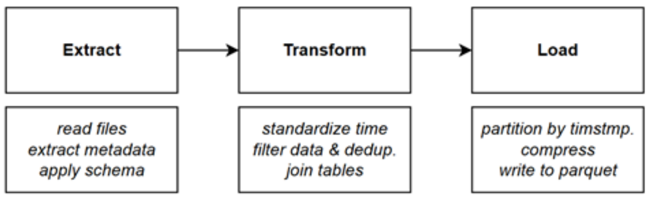
Task 2.3

Our approach assumes that a cancellation is relevant if it affects either the arrival or the departure of a scheduled stop. To accurately capture this, we performed left joins between our central fact table and the time dimension table. We used left joins to ensure that we did not lose records where only one part of the stop contained a cancellation timestamp. We also utilized a cross join with a temporary time window table to bring the snapshot parameters into the query's scope.

Task 2.4

We made some assumptions for this task. First, we assumed that "delay" specifically refers to departures, as these have a more direct impact on passenger travel planning than arrivals. Second, we implemented the assumption that only positive delays should be included in the average. We argue that “negative delays” should not be used to mathematically offset or decrease the average delay. Finally, we remained case-insensitive for the station name to be consistent.

Task 3



If the data we intend to extract, transform, and load fits easily within a single machine’s memory, standard Python or similar local approaches are often sufficient. However, for large scale data processing, Spark is often useful, as it preads the workload across a cluster of nodes.

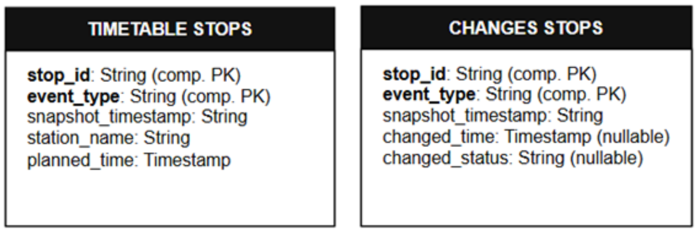
Task 3.1 - ETL Pipeline

1. Extract

Spark operates via a master-worker hierarchy. The master (/driver), serves as the central orchestrator of the application. The workers, meanwhile, provide computational resources. On these workers, we assign Executors, which are distributed processes responsible for executing the code assigned by the Driver. They do their work on the CPU cores that we assign, in parallel. Since our current implementation is limited to a single machine rather than a large cluster, we ran the tasks with the master only (local mode), to optimize time. This is because switching data between separate containers workers is computationally expensive due to serialization and network overhead, especially considering the number of files. By running locally while still assigning multiple CPU cores, we maintain parallel processing capabilities while reducing the overhead. To further minimize latency, we extracted our tar files directly into the local environment instead of relying on a shared mount, effectively eliminating the time required for file transfers. While this specific approach would be counterintuitive in an industrial setting with more resources, it speeds up local execution.

When reading our XML files, we initially implemented a binary file reading approach. This method involves reading the raw bytes of the files and using standard Python libraries to parse the content. While functional, this approach forced us to rely on native Python execution and iterative for loops to handle data input. This created a bottleneck for computation. To overcome this, we switched to spark-xml, an external package specifically designed for Spark. We provided spark-xml with explicit, custom schemas to reduce overhead while scanning files.

We employ two distinct methods for reading our XML schemas. One is for the timetable files and one for the change files. In both instances, we only read the attributes necessary for the analytical queries in tasks 3.2 and 3.3 to minimize memory consumption. Our tables are designed to represent a single stop event. You can see an outline of our schema for these two intermediary tables below.



TIMETABLE STOPS
stop_id: String (comp. PK)
event_type: String (comp. PK)
snapshot_timestamp: String
station_name: String
planned_time: Timestamp

CHANGES STOPS
stop_id: String (comp. PK)
event_type: String (comp. PK)
snapshot_timestamp: String
changed_time: Timestamp (nullable)
changed_status: String (nullable)

Event Type refers to arrival/departure. The other names are self-explanatory. It is noteworthy that we have a composite Primary key as a stop can be either arrival or departure. Also changed_time and changed_status are nullable, because a cancelled train might not have a changed time, and we don't need changed_status for queries. We included the changed_status attribute to identify cancelled trains for later analysis. We defined all fields during the reading process as nullable though because Spark's schema enforcement is strict.

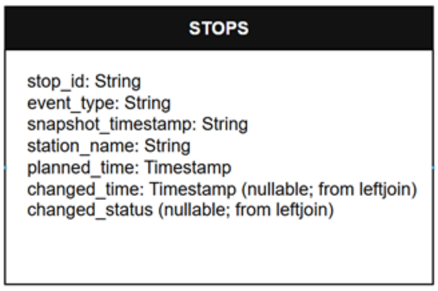
2. Transform

We then perform a union over the arrivals and departures. To ensure the integrity of result, we strictly remove any records that lack a stop_id, a planned time, or a station name.

Once filtered, we performed a secondary analysis to detect anomalies. We specifically investigated whether spelling variations in station names existed (e.g., "Berlin Hbf" vs. "Berlin Hauptbahnhof"), which would artificially inflate the station count, though no such errors were found. Additionally, we validated the uniqueness of the stop_id. Our analysis revealed that the ID field is not inherently unique per event type as initially expected. To resolve this, we implemented a majority vote deduplication strategy, similar to the approach in Task 1. By grouping by all attributes and selecting the most frequent occurrence, we mitigate the impact of conflicting data entries. Also, by standardizing timestamps to a compatible format at this stage, we ensure the data is ready for analytical tasks. We chose not to transform the station names at this point, but rather to implement a matching algorithm at query level, to save some computation time on our device.

While we explored the concept of lineage in the lecture, we did not include it in our schema, upon asking in the forum. It would have been an additional interesting consideration for this topic. It is also important to note an assumption of our current approach. If a train appears exclusively in a change file as an added row, it will be excluded from the final dataset, as our pipeline relies on the Timetable as the primary source of truth. Thereafter, we proceed with combining these two tables into a single, unified dataset. Since change files frequently contain multiple entries for the same stop_id and event_type, we first apply a filter to retain only the most recent event for each record.

Once both tables are prepared, we perform a left join over the stop_id and event_type. It ensures that all planned arrivals and departures from the master timetable are preserved, even if they do not have a corresponding delay or change entry in the change files. During the merging process, we selectively integrate the changed_time and changed_status from the change records. Additionally, in instances where a delay occurs, we refer to the snapshot_timestamp from the change file to gain a clearer insight into exactly when the train arrived / departed. The resulting schema is presented here:


STOPS

stop_id: String
event_type: String
snapshot_timestamp: String
station_name: String
planned_time: Timestamp
changed_time: Timestamp (nullable; from leftjoin)
changed_status (nullable; from leftjoin)

We chose this flat schema to leverage Spark's full potential for analytical processing. By employing a wide, tabular format, we take advantage of columnar storage in Parquet, which allows Spark to read only the specific columns needed for a query. This design is particularly effective for the analytical queries in tasks 3.2 and 3.3, as it enables aggregations and filtering without the overhead of repeated data parsing or complex joins during the analysis phase.

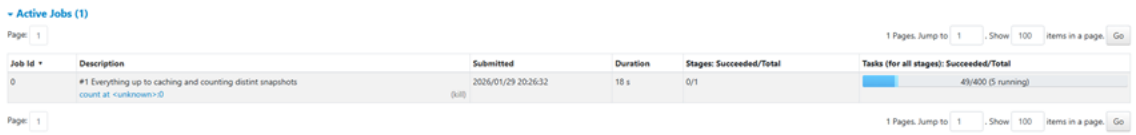
3. Load

As a final step, we wrote the resulting Spark DataFrame to the Parquet format. Parquet stores data column by column, which allows Spark to prune columns and only read the specific fields required for a query. Furthermore, Parquet is a distributed file format. It splits data into multiple files that can be stored across different nodes in a cluster, enabling Spark's workers to read these files in parallel and significantly increasing I/O throughput in cluster mode. We chose to save these files using Snappy compression. Snappy is a compression and decompression library that aims for very high speeds and reasonable compression ratios rather than maximum compression.

Pipeline Optimization

Throughout the pipeline, we aimed to maximize Spark's distributed computing capabilities while managing the physical constraints of our hardware. In the initial configuration, we enabled Adaptive Query Execution. This allows Spark to optimize query plans at runtime based on the actual statistics of the data being processed. Alongside this, we enabled Coalesce Partitions, which permits Spark to automatically merge small partitions during the execution of a shuffle to avoid unnecessary overhead.

However, we also manually applied coalesce at a specific point in the ingestion process. After reading the raw files with spark-xml, we noticed the creation of an excessive number of partitions. Managing thousands of tiny partitions created and overhead, as the Driver had to coordinate many tasks. By forcing Spark to coalesce these into 200 partitions, we therefore speed up the transformation phase. We can gain insight into how Spark organizes these jobs by looking at the SparkUI. For each job we added Job descriptions to give more insights and to debug:


Active Jobs (1)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	#1 Everything up to caching and counting distint snapshots count at <unknown>:0 (kill)	2026/01/29 20:26:32	18 s	0/1	49/400 (5 running)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Another optimization was performed before the Load phase. Immediately prior to writing the data, we calculate the number of distinct snapshot timestamps and use repartition to match that count based on the snapshot attribute. Since we aim for our Parquet output to be physically partitioned by snapshot_timestamp, this step ensures that each Executor/Core handles all rows for a specific snapshot in a single block. Without this repartitioning, they might attempt to write different parts of the same snapshot simultaneously, resulting in multiple tiny Parquet files being created for a single time snapshot_timestamp. This improves writing time and query time later on

The final optimization we did was to cache our joined DataFrame after merging timetables and changes. This is because up to this point, the spark operations we used were all lazy, meaning they were not executed but rather waiting for execution by an active command, such as count(). When we trigger an active command spark executes these steps. Now, if we didn’t use cache, the next active command would have had to recompute these steps all over again.

Task 3.2 - Average Delay Query

Spark offers an advantage in querying large datasets through predicate pushdown and partition pruning. It leverages the metadata within Parquet files to skip irrelevant data blocks entirely. By partitioning our dataset by snapshots, we allow the engine to physically narrow its search to specific time windows. We begin by reading the Parquet dataset into a Spark session. A common challenge in transit data is that station names provided in a query may not exactly match the database records due to typos or different naming conventions. To solve this, we implemented a string matcher utilizing Levenshtein distance. This algorithm calculates the edit distance between strings to find the closest approximate match in our station list. To increase the robustness of this match, we strip common, noise-heavy appendices like “Berlin” before comparison. While this matching is generally robust, it leaves some error and differs from our approach in Task 1 as we wanted to try out different approaches. Similarly, if the delay is negative, it still counts towards our query, in contrast to Task 1. The same holds for other assumptions. It would have been possible to also only count departures here, but we wanted to experiment with different approaches to the solution to demonstrate our different approaches to problem solving.

We then filter out cancelled trains. We also decide for the changed_time to take precedence over the planned_time when it comes to deciding the day, e.g. for trains departing after midnight due to a change. This ensures the delay is attributed to the day it actually occurred rather than the day it was scheduled. We calculate the delay for every individual arrival and departure event and then aggregate these into daily averages. We intentionally exclude days with zero movements from our final average. This would artificially inflate the station's true performance. Therefore, our result represents the average of the daily averages for only those days when the station was actually operational.

This method of evaluation could lead to Simpson’s Paradox as explained in the lecture, where the average delay of a station over a month might appear better than its daily performance suggests, because we ignore the weights of each daily average.

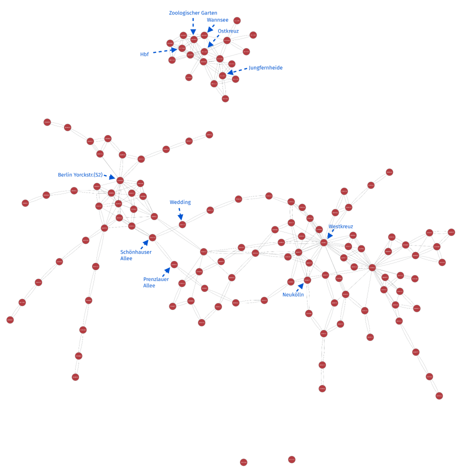
Task 3.3 - Average Peak Hour Departures Query

For Task 3.3, we initialized a Spark session and accessed the Parquet dataset to determine the average frequency of train departures during peak hours for each station. Our understanding of this task is to give out the average number of departures at peak hours for every station, therefore resulting in a long output. In this instance, a day without departures does hold relevant information to our desired task context. We therefore start by counting the total number of distinct dates in our parquet dataset, leading us to the length of the evaluation period by which we can later divide by (if a station didn’t have a departure for a given day during peak hours). We then filter for departures and filter out cancelled trains. Again, even though the rush hours are far from midnight, we rely on changed_time as the chosen field for deciding the time. We then extract the departure hour from that field and filter for it. Eventually, we count the number of departures per station and divide by the number of total dates to receive our results.

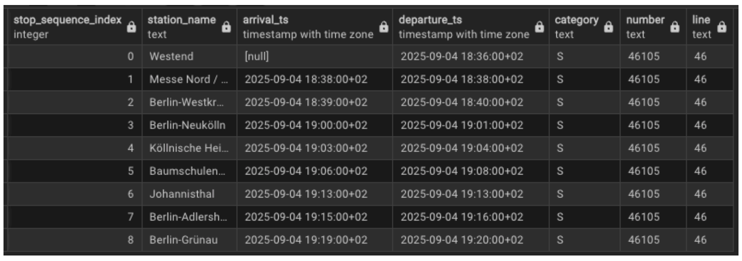
Task 4

The ETL pipeline developed in task 1 can be reused to a great extent in task 4. Key challenge was expanding the star schema with the stop order per line. Also, timetable changes were ignored for notebook performance reasons and due to not being needed in the tasks.

The information on stop orders and therefore on edges connecting the different stations of the planned graph, is only available in the timetable files and is not consistent across different times for the same line (e.g. partial trips). We have considered hard-coding the train routes as transportation systems work with timetables as long-term artifacts but recognize that including disruptions (such as constructions etc., unspecified for the given duration) call for continuous maintenance. Another option to derive the longest route by majority vote from the available represented routes and adding sequence indices per stop, mistakenly builds on historical facts that might be updated and lead to instability. It was therefore decided to include a stop sequence index that references the specific train run (line and service day) belonging to the stop and therefore avoids instability. In order to correctly identify stop order sequences that can be used to infer edges for the classic transport network (stations as nodes) and time-expanded graph, edges can only be drawn where connected train runs are identified. Using the star schema, this identification works via train_id per day, and to catch trains running through midnight, train runs are identified by train_id in a 4h interval.



As the data contains several mistakes, especially missing S-Bahn trains at big stations, the presented stable logic leads to ”missing” edges when comparing with potential edges found in the ppth attributes. This is accepted as a data source issue that doesn’t interfere with algorithmic correctness (Forum: Missing S-Bahn Data). For example, the forum example asking for the shortest path between Hbf and Tiergarten will be answered here as “no paths” as Hbf and other stations serving REs, RBs, and others are not connected with S-Bahn stations. For better understanding of station connectedness derived from the data, a graph belonging to task 4.1 is attached here and annotated with a view station names.
Busses are excluded for both graph building tasks as discussed in the forum, as they include stops not referenced in our station table and lead to unusual connections between S-Bahn stations that confuse 4a’s shortest path search (e.g., skipping Ringbahn stations). As our station table builds on the stations json, only the stops of a stop order that belong to the Berlin area will be considered for the graphs.
Some sanity checks were made, e.g. Neukölln and Westkreuz are connected in the graph but usually not by a train line - still, this is represented in the data and therefore true regarding our graph building rules (see train run represented below).



stop_sequence_index integer	station_name text	arrival_ts timestamp with time zone	departure_ts timestamp with time zone	category text	number text	line text
0	Westend	[null]	2025-09-04 18:36:00+02	S	46105	46
1	Messe Nord / ...	2025-09-04 18:38:00+02	2025-09-04 18:38:00+02	S	46105	46
2	Berlin-Westkr...	2025-09-04 18:39:00+02	2025-09-04 18:40:00+02	S	46105	46
3	Berlin-Neukölln	2025-09-04 19:00:00+02	2025-09-04 19:01:00+02	S	46105	46
4	Köllnische Hei...	2025-09-04 19:03:00+02	2025-09-04 19:04:00+02	S	46105	46
5	Baumschulen...	2025-09-04 19:06:00+02	2025-09-04 19:08:00+02	S	46105	46
6	Johannisthal	2025-09-04 19:13:00+02	2025-09-04 19:13:00+02	S	46105	46
7	Berlin-Adlersh...	2025-09-04 19:15:00+02	2025-09-04 19:16:00+02	S	46105	46
8	Berlin-Grünau	2025-09-04 19:19:00+02	2025-09-04 19:20:00+02	S	46105	46

Task 4.1: Shortest path

The graph creation is based on the connection assumptions and rules explained above and therefore represent all possible conditions present in the available time duration as described. More in detail, the graph represents each station s as a node. For each connection c between two stations s_0 and s_1 (represented in the stop sequences that can be derived from combining train number, service day, and stop_sequence_number), an directed edge is added between the two stations, if none is already existing. After testing on undirected edges that might represent ideal conditions of network connectedness, a decision was made to work with directed edges (following the planned stop orders) which better represent transport network logic.The graph is unweighted, as we are only looking to count graph hops and don't consider distances.

We chose to use the `graph-tool` library which builds on C++ as it "can be orders of magnitude faster than Python-only alternatives, and therefore it is specially suited for large-scale network analysis" (https://graph-tool.skewed.de/static/docs/stable/index.html). graph-tool automatically uses breadth-first search for unweighted graphs, Dijkstra's algorithm for positive weights and the Bellman-Ford-Algorithm for negative weights.
Example stations are pre-chosen in the notebook (and can be replaced). The shortest paths in between them in terms of graph hops will then be computed.

Task 4.2: Earliest Arrival

The graph is built on the conditions described above: connectedness is derived from stop order sequences, representing planned train routes, that can be derived from train_id, service time window, and stop_sequence_index from the fact table.
We chose building a time-expanded graph that can work with an extended Dijkstra algorithm to find the earliest arrival time. References to this common strategy can be found in Dorothea Wagner's lecture "Algorithmen für Routenplanung" (KIT, 2019) and John Miller's APSP script (Stanford, 2014).
The time-expanded graph is characterized by having one node per station s. Additionally, for each two connected stations s_0 and s_1, another node is added for each trip t connecting s_0 and s_1. An s_0 -> t edge is added and assigned as weight the planned departure at station s_0, edge t -> s_1 is assigned as weight the planned arrival at station s_1.
This makes sure that only trips that were actually planned in the given two weeks are added as edges to the graph, and therefore the algorithm will only find paths that include time information.
The Dijkstra algorithm is extended to accommodate searching for the earliest arrival, so the edge with the earliest arrival time that connects a trip t_i with the goal station s_to, forming a path beginning at station s_from. As Dijkstra requires ordered, numeric edges, the departure and arrival timestamps are converted to integer epoch second. Weights are then not added, but earliest arrival time is updated per node.

Please note: As bus trips are ignored, some potential replacement services are missing that might lead to confusion. E.g., trips from Schönhauser Allee or Prenzlauer Allee towards S41 Ringbahn stations are not possible before September 8th, leaving a missing service window of six days.