Thoughts about Transit Data

Arttu Kosonen

20.12.2021

Contents

Introduction

This is a series of texts where I consider some aspects of public transport data modelling, based on my personal experience writing my Master's thesis from year 2019 on, and working at HSL (Helsinki Region Transport) as transit planner and data engineer. In the following sections, I will discuss my thoughts about time and place representation in transit data, and what software tools I have used and like to recommend for dealing with transit data.

These texts represent my own thoughts and experiences - not exhaustive, scientifically proved facts or common best practices. There is no answer how to best model and use transit data, and what tools to use for it - it depends! And I've only just started my own path in the fascinating world of transit data modelling.

Estimated time use

For the 5 ECTS that I wish to acquire from Aalto University text collection, I have calculated a *rough* estimate of time I've spent with topics related to these texts. These hours come, in principle, on top of my paid working time and time spent to my Master's thesis, although they all have to do with the same topics.

Topic	Description	Hours
Time and place in transit data	Identification of temporal and spatial attributes, their units, and relationships in transit planning and data models. Explored, tested and learned largely during Spring 2020, as I developed a schedule data model as part of my thesis, but the schedule model part was eventually dropped from the final thesis. In 2021, further learning in the Jore4 project at HSL.	20
GTFS schedule & network data	Exploring the GTFS data model and open HSL GTFS datasets since 2019 when preparing for the thesis. Later on, many kinds of GTFS applications in analytics at HSL, due to the limited access to original planning data in Jore3.	30
HFP operations data	Core of my thesis: a lot of trial-and-error work with large amounts of HFP data since 2019, such as finding the best ways to format, store and query the data, and possible types of errors. Most of the hands-on work not visible in the thesis.	40
Data validation	Conceptualization and proof-of-concept testing of data "content" quality monitoring at HSL since end of 2020, as it was recognized that monitoring only the health of technical systems is not enough, and there is a need for a feedback loop of monitoring and actions to improve the reliability of both transit planning and operations data.	20
Data tools	Learning about data and coding related tools as part of everyday work and studies. Especially software library documentation, vignettes, and blog posts about technologies, techniques, workflows, and visualization of relational, spatiotemporal and movement data.	30

1 Time in transit data

Collective transport seems like a relatively simple thing: buses and other vehicles run from stop to stop, at scheduled clock times that pretty much remain the same from day to day. Sometimes they run a bit ahead or late from the schedule. In reality however, the temporal and spatial aspects of a transit system are more complicated than that. I'll discuss my views of the temporal aspects in transit data modelling in this post, and the spatial aspects in the next post, Chapter 2.

I'll be using the Finnish date format like `1.12.2021` in this text. It follows the order `day.month.year`.
Clock times, in turn, are expressed here like `11:35 (HH:MM)`, `11:35:14 (HH:MM:SS)`, or `11:35:14.967 (HH:MM:SS.sss)`.
Everything is handled in local Finnish time, unless stated otherwise.

I'll be giving some tips for data modelling from my limited experience. They are best applicable to the PostgreSQL database environment, but similar concepts can of course be found in other environments too, such as R and Python.

Time is an overloaded term - it has many meanings and uses. In our organizational language at HSL, as well as elsewhere, each meaning is most often well understood from the context. At the same time, there are many caveats and possibilities of at least slight misunderstandings with the concept. Especially, in my view, when an experienced transit planner or analyst discusses our everyday business and related data models with a "newcomer", such as a software developer still onboarding a project, it typically takes a lot of time to unpack the exact definition of time in each context and to get both on the same page. It often feels overwhelming for a newcomer to read all the meanings from a list of standards, for instance, and on the other hand, trying to skip that by simplifying things and making assumptions by "common sense" just seems to postpone the inevitable effort of confronting the complexity of time in transit modelling to a later phase. At least at HSL, many essential definitions of time aspects - overlapping or contradictory ones as well - have not been thoroughly documented and officially agreed upon, which makes learning them pretty hard. I think this is because most experts already share a similar mental model about them, and are able to use a common language understandable enough, and all of these have evolved over the years; in the end, for an expert like this, it feels like there is always much more critical work to do than write down such "self-evident" things. But then, every time we start a development project with external non-transit experts, or hire a new planner or analyst and wish to onboard them to the everyday work, we have to confront the fact that we have a whole lot of undocumented knowledge like this, and even the experienced ones do not in fact share *exactly* similar mental models.

Well, I try to contribute to explaining the dimensions of time in transit planning and modelling a bit. My views are anything but exhaustive, though.

Let's think of a transit trip of HSL bus line 23, starting at 12:23 at Central Railway station on 11.12.2021. It has a schedule - planned departure, possibly even separate arrival times, at each stop - and hopefully, a vehicle can realize the scheduled service at real times that follow the planned times within a satisfactory variation. What aspects of time are involved, and where do they originate from?

1.1 Times, durations, deviations

On that very day, both the scheduled and realized stop times can be thought of as *absolute timestamps*. That is, they have a date and a clock time part, and they happen in a time zone, namely `Europe/Helsinki`,

or `GMT+02`, in this case. This means we could also convert those timestamps to *UTC time*, and in fact, computers do that most of the time - they store absolute timestamps as decimal seconds starting from the so-called UNIX epoch, UTC midnight of 1.1.1970. For instance, the departure time `11.12.2021 12:23:00` would be stored as `1639218180.0` UNIX seconds.

However, planners do not create absolute timestamps for scheduled trips by hand. They are generated otherwise.

First of all, a transit route during an operation season (such as winter 2021/22 or summer 2022) requires *run times* for different basic day types. Typically, the run times are created separately for three weekday types: Mon-Fri, Sat, and Sun. This is because the real run time patterns from stop to stop throughout the day tend to vary most between these three classes, but not so much between individual working days, for instance. The COVID-19 pandemic has probably changed this a bit, though, as there seem to be more random, interesting travel peaks on certain individual weekdays, which makes it a bit harder than before to plan reliable run times.

A run time means the *duration* it takes to operate from a stop to another stop. Duration is a continuous value, typically measured in decimal minutes or seconds. The latter is most common in computer systems, and it can be easily decomposed into components for better readability:

```
dur_s <- 324.97
sprintf('%02d:%02d:%.3f',
        dur_s %/% (60*60), # hours
        dur_s %/% 60,      # minutes
        dur_s %% 60        # seconds
)
```

```
## [1] "00:05:24.970"
```

Note that the duration seems like a clock time, but it is not bound to any point in time, unless such a context is actually provided. In the context of run time planning, durations of a stop-to-stop segment are bound to time of day, since real run times tend to vary during the day due to various reasons, such as other traffic, more or less passengers being served at stops, and so on. Thus, the *run time depends on the time of day*. Wouldn't it be practical if we didn't use two meanings of *time* in the same sentence, and could instead speak of *duration* when meaning the run time…? Well, that is against what feels common and familiar.

Actually, it makes no sense to plan run times for *every* stop along a route. In the big picture, run times tend to behave in a linear manner over multiple adjacent stops, as long as the traffic environment and other affecting factors remain pretty similar. Moreover, variability of planned vs. real stop times of less than a minute is considered insignificant at least in street transport (bus & tram in HSL area): although real run time *measurements* have the accuracy of seconds, planned run times are defined in integer minutes, since the public schedules are shown at the same accuracy, and it would not be worth the effort to create run times and schedules at the precision of a second. (In modes of a higher right-of-way, such as metro and train, this makes more sense, since they can better adhere to schedules with margins of seconds rather than minutes, as there less external factors disturbing them.) Instead of individual stops, we plan run times for segments of multiple stops, i.e., between *places*: a stop is assigned a *place* label, if there's a change in the run time behaviour significant enough so it makes sense to start a new segment for separate run times. Typically this can happen when moving from traffic light filled urban area to a highway, or whenever a stop is defined as *timing stop*, i.e., the vehicle must wait until the scheduled departure time before proceeding again. Read more about stop places in Chapter 2.

The run time of a *segment* between places is "spanned" over all the stops belonging to the segment, and stops in between thus get linearly interpolated duration values calculated from the beginning. If the segment run time is short enough and stops are close to each other, then adjacent stops might get seemingly same duration values, and eventually the same departure and arrival times, since the planned time values are

coerced to whole minutes. Keep this in mind when examining the public stop schedules or wrangling GTFS stop times, for instance.

After run times of a route have been created for a season and the basic day types, it's time to create the schedules. In the schedule of a route (and direction - typically a route has two direction variants), we have *trips* that follow the pattern of successive stops. To achieve *clock times* at stops for each trip, we need two things: firstly, an *initial start time* of the trip from the *first stop*, and secondly, the *run times* for the pattern of stops that we just created. Clock times are then produced by simple addition of the initial start time and cumulative run times:

Stop	Run time	Cumulative run time	Departure time
Stop 1	0 min	0 min	12:00 (initial)
Stop 2	2 min	2 min	12:02 = 12:00 + 2 min
Stop 3	1 min	3 min	12:03 = 12:00 + 3 min
...	...	...	...
Stop 1	0 min	0 min	16:00 (initial)
Stop 2	3 min	3 min	16:03 = 16:00 + 3 min
Stop 3	2 min	5 min	16:05 = 16:00 + 5 min

Although at HSL we tend to generate the stop times like above, using the first stop as a reference, it is possible to do it other way around too, calculating the times "backwards" from the last stop, once that last arrival time has been fixed to a desired clock time value. That would make sense if it is important that the trips arrive accurately to a train station to allow reliable transfers from bus to train, for instance. However, "initial start time + run times" is the default method. Moreover, a planner can add extra time for individual trips at certain stops by hand, if needed: this can sometimes be an easier way to improve the schedule adherence of individual trips than going through the run times -> entire schedules process again.

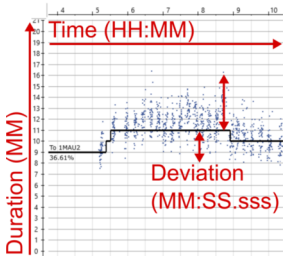

Figure 1: Example runtime analysis from Hastus. Three different dimensions of time annotated.

Figure 1 is a screenshot from Hastus run time analysis and planning tool, where the small dots show measured run times on a stop-to-stop segments, and the black line shows a fitted run time *plan* that can vary during the day. The planned run time is defined in integer minutes and saved to the Hastus database so it is then used as a basis for schedules of the day type in question, and considering time of the day, depending when a scheduled trip passes the segment. The percentage value shows the proportion of observed run times that are shorter or equal to the planned run time: i.e., in the example, 37 % of the trips would have passed through the section on time or faster.

I have indicated three different aspects of time that can be found in Figure 1:

1) Time of day, or clock time, is presented on the X axis. Run time is dependent on it - not inherently, but due to typical time of day bound phenomena, such as volumes of other traffic.
2) Run time can be seen on the Y axis, in minutes. As indicated, I'd like to call it *duration* to not confuse it with *time* as *clock time*.
3) *Deviations* of duration can be seen on Y axis as well, when comparing observed and planned run times. In general, a deviation could be calculated as `duration +- duration`, `time +- duration`, `duration +- time`, or `time +-time`. Even though it can be measured with same units as duration - such as in minutes, or in case of observations here, precisely in minutes, seconds and milliseconds - I think

deviation is conceptually different from duration. Whereas duration describes how long an actual event takes, how long a feature is valid, etc., deviation describes the *time difference* of instants or durations of comparable events. I'll discuss duration a bit more in Part 1.3.

1.2 Operating days and 30-hour clock

The concept of *date* in a given time zone is a pretty simple one: a new date, `+1 day` to the previous one, begins every midnight at 00:00. Unfortunately in transit scheduling, it's not as simple as that. Consider a trip that starts in late Friday evening and has a schedule like this:

```
stop_1 23:49
stop_2 23:54
stop_3 23:59
stop_4 00:01
stop_5 00:05
```

In schedule planning, every regular trip originates from a vehicle schedule that is required to have a *day type* attribute: usually the day type can be something like `Mon-Fri`, `Sat`, `Sun`, or a special day type, such as `Independence Day`. The day type is used to assign the trips of a vehicle schedule to real calendar dates, i.e., an identical set of trips of a `Mon-Fri` vehicle schedule is assigned to every working day during the season, such as our current winter operation season 16.8.2021-19.6.2022. This way we don't need to assign the trips manually to each date, and operators and passengers get schedules that are easy to understand, as well. (Day types can override each other, meaning in practice that if `Independence Day` is assigned to a normal Monday, then that special day type is prioritized instead of `Mon-Fri`, except if a vehicle schedule has no separate version specified for that special day type, then it still follows `Mon-Fri` trips. I'll not go deeper into this here.)

Now, you see that the calendar date clearly changes after the third stop in the example above. Still, the example trip must be assigned to one vehicle schedule and day type exactly - we cannot split it into two trips, as it must be a continuous trip run with the same vehicle, from both operator's and passengers' perspective. And our planning software (Hastus) obviously does not allow "jumping" backwards in the timeline during the trip.

The practical answer is *operating day*. The main point is that *an operating day spans over the real midnight*, allowing trips to be partially or entirely assigned to the day type that in reality refers to the previous calendar date. In vehicle schedule planning, it means a timeline of a day like this:

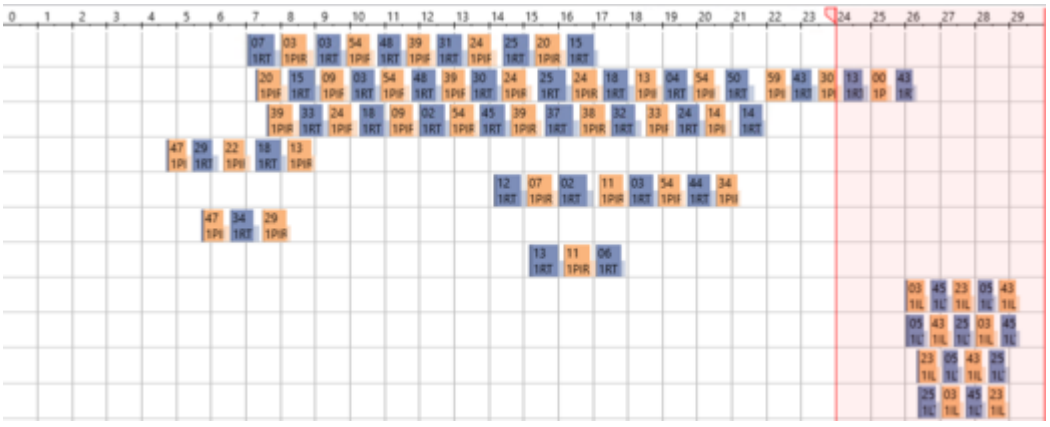

Figure 2: Example of 30-hour timeline in Hastus. Trips after 24:00 get will actually start after 00:00.

Hastus uses the definition of 30-hour operating day and therefore allows trips of the "previous day" to last until 6:00 on the next real day. Figure 2 shows a practical example of this: a block graph, where successive

trips are assigned to "blocks" (set of trips for a vehicle), and the times of an operating day are presented on a 30-hour timeline. HSL, on the other hand, tends to define that an operating day starts at 0:00 and end at 4:30 on the next calendar day: this is due to practical reasons, as there is usually as little traffic as possible going on just between 4 and 5 in the morning. But this is more of a practice than an exhaustive definition, and it is not enforced technically, as far as I know: any trip that is valid in Hastus' timeline will do. 30-hour operating day is also used by GTFS, so I think it's a reasonable default length to assume when considering transit times in data models, unless stated otherwise.

In addition to stretching the timeline needed for the trips of a day type a bit, there is another benefit from operating days: entire trips can be defined after 24:00, if they are only required on Friday or Saturday nights, or night before a holiday. Although it would be technically OK to define them on the next calendar day after 00:00, this would not be logical to the passengers: 30-hour operating day allows such trips to be visible in the public schedule tables under Friday, Saturday, and so on, not "randomly" under the next weekday. In Figure 3, stop times of line 23 after 0:00 have originally been modelled as `> 24:00` times, this way they are kept under the Mon-Fri day type and not shifted to the next visible day type, Sat.

22.43	22.53	22.29	22.52
23.13	23.23	23.00	23.22
23.43	23.53	23.30	23.50
0.13	0.22	0.00	0.19
0.43	0.51	0.30	0.49
1.13	1.21	1.00	1.19
1.43	1.51		

La / Lö		**La** / Lö	
Rautatientori Järnvägstorget	> Porvoonkatu Borgågatan	Pirkkolan urheilup. Britas idrottspark	> Pasilan asema Böle station
5.27	5.35	5.13	5.33
5.57	6.05	5.43	6.03
6.27	6.36	6.13	6.34
6.57	7.07	6.43	7.04
7.27	7.37	7.13	7.34

Figure 3: Example of public schedule of line 23 in night time. Screenshot from reittiopas.hsl.fi.

Obviously, operating days and calendar days go hand in hand, as long as both start in the midnight, but then there's an overlap in the night: the time range `00:00-06:00` of a calendar day `5.3.2021` equals both `00:00-06:00` of `5.3.2021` and `24:00-30:00` of `4.3.2021` measured as operating days. We can thus note that the conversion from operating days and 30-hour times to real timestamps is irreversible for times between 00:00 and 06:00. This introduces a challenge in data modelling: whether to express days and times as operating days and 30-hour times that represent well the planning context of transit, or as real datetimes/timestamps that are technically unambiguous, since they refer to exactly one point in the real timeline. Based on my own experience, I'd avoid this caveat by using operating days and 30-hour times as far as possible on the *planning side*, and keep using real datetimes whenever *real operations*, such as operated trips, are modelled - since real events tend to happen in a continuous, unambiguous time horizon.

One more note to make the concept of operating day not too simple: GTFS Reference does not refer to operating days but *service days* - which makes as perfectly much sense, too, in my view. This is just one of the many examples how in transit modelling, a term can often have synonyms, none of which is necessarily the absolutely correct one to use. The dominating one often tends to be determined by traditions and practices of organizational language, and I haven't seen a standard document either that would manage to

cover all such cases.

One more, even more important note from GTFS: I only just learned that Google tells there that *the time is measured from "noon minus 12h"*. Why? Because if you want to take *daylight savings* into account correctly, you cannot actually use midnight as the reference, as timezone shifts tend to be done at 3:00 or 4:00 in the morning, at least in Finland. Using the midnight reference would then mess up matching calendar and operating days on those dates when the shifts happen. This new information blowed my mind, and I still need some time to refresh my conceptions and opinions on how to correctly model transit date and time data on the planning side…

1.3 From planned to real times

On the planning side, we first defined clock times (such as trip stop departure times) for vehicle schedules specific to generic day types, which we then converted into clock times per date, i.e., datetime values by using the information on what calendar dates correspond to the day types during the operation season. Right after we generated the run times using seconds-precise run time measurements from past real transit trips, we have in practice operated at the precision of integer minutes: i.e., with clock times like `11:03`. Our systems would allow operating with seconds as well, and this is actually done when planning schedules for the subway mode, but in bus and tram planning, trying to determine things beforehand at that level of detail would make little sense.

The temporal dimensions in the planning domain are packed with human-centered time conceptions, like "calendar", "special days", "day of week", "rush hour", and with time values coerced to limited precision such as minute and whole date. As soon as we move on to the real world - *"realization context"* - things start to actually happen at exact time instants that can be put on a continuous timeline, measured with concepts like UTC Unix time. I think this conceptual difference is good to understand when creating data models specific to each context, and especially when integrating these two contexts. Our systems and software tools provide us with great methods for coercing time values, "bucketing" them into classes like hour of day or quarter of the year, taking time zones into account, and so on. These all are widely required when matching planned operations to realized ones and vice versa. In my experience, the main part that often fails there is the human actor: we tend, again, to oversimplify things at intermediate steps so they "look and feel nice" - this often means things like dropping time zone attributes, coercing realized timestamps to integer minutes, or exporting planned transit trip times in 24-hour clock system so they are in the same scale with real 24-hour times, although this conversion ultimately results in loss of information, like I mentioned above. There's probably no silver bullet to this topic either, but I think it helps a lot if we're able to identify the domain we're dealing with in each case, take the time to really consider whether accurate UTC times or coerced "generic" time components are the right answer - and thoroughly document these thoughts and design decisions along with the data model we create. In this fields, misunderstandings and wrong assumptions are likely to happen otherwise.

Time *deviation* is an especially useful concept for comparisons between planned and realized transit times - run times (durations), clock times, or both of these. Many performance measures in transit analysis are based on schedule adherence, which requires comparison between planned and realized stop departure and arrival times, for instance. In such comparisons, different precisions of the planning and realization domains must be considered: for instance, if stop times were planned with whole minutes, it does not make sense to consider a deviation of `+24 s` as real delay, at least a significant one. For these situations, a strategy should be developed for dealing with different precisions. Sometimes it may mean rounding, sometimes using `floor` and `ceiling` functions, for example.

Deviations are useful not only between these domains but also within them. I would for instance think "headway" as one kind of a deviation - the difference between successive trips. Headway is, of course, measured either within planned or realized trips. (From passenger's perspective, headway could rather be a *duration* - how *long it takes* to wait for the next vehicle.)

Deviations are most practically measured in hours, minutes, and / or seconds, and stored as decimal seconds internally. They are technically similar to durations, but conceptually, in my view, *duration* measures a

different thing - it focuses on an event, rather than the difference of two comparable events. This is however a matter of taste and interpretation to some extent: for instance, we could as well think that two clock times at successive stops are "comparable events", even though they result in a run time describing the stop-to-stop "event". With this in mind, I'm not able to give exhaustive definitions for my conceptions of *duration* and *deviation*.

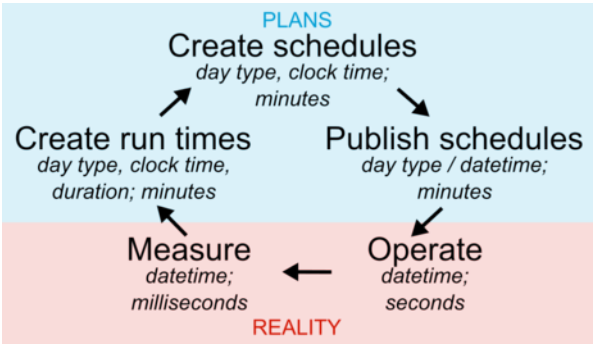

Figure 4: Feedback loop of transit planning and reality. Aspects and precision of time in each step.

Figure 4 summarises the connection between planned and real transit times, and demonstrates two things:

- firstly, how the real, actually operated transit service affects the planning through measured run times, and how planning then again creates the basis for real transit operations and times through schedules;
- secondly, what aspects of time are applied at each phase of this loop, and at which precision.

Tips for data modelling

- In planning context, keep using operating days and 30-hour times if you possibly can: they can be converted into real datetimes unambiguously, but not vice versa.
- In real operations context, it is wise to use real datetimes, otherwise you can easily mix up whether an event happened today or yesterday night. Remember that you can still, for example, easily match a real trip to the planned one by using the conversion from operating to real days mentioned above:

```
Planned trip starts at 4.5.2020 25:01:00 (operating day)
Real trip started at 5.5.2020 25:02:03 (real day)
> Set these to the same timeline by converting the planned trip:
  date 2020-05-04 + 25 hours + 1 minute + 0 seconds == 5.5.2020 01:01:00
```

- In Postgres, do not try to use the `TIME` (24-hour clock time) type for 30-hour operating times. Instead, use `INTERVAL` that makes it easy to store time values starting from midnight as decimal seconds while representing them as `HH:MM:SS`. Moreover, addition `DATE + INTERVAL` results directly to a correct `TIMESTAMP` value, and of course takes into account `> 24:00:00` times, resulting in a timestamp on the next calendar date. Just remember to add the correct timezone information when converting to `TIMESTAMP WITH TIME ZONE`!
- In R when using packages such as `{lubridate}` or `{hms}`, as well as Python, conventional time representations will not support 30-hour clock times either. Find something comparable to the Postgres' `INTERVAL` type - probably you'll have to mess around with `Date` and `difftime` types together in R. I've often stored the times as plain decimal seconds from midnight, and used a custom formatter function that can render the values as 24+ hour `HH:MM:SS(.sss)` labels, when creating plots with a 30-hour timeline, for instance.

1.4 Human times

Transit planning is full of time concepts that have nothing to do with time as a *physical* phenomenon while they are extremely important to *humans*. I mean things like "calendar day", "week", "rush time from 7 to 9 in the morning", and so on. Knowing no official or otherwise better term for these, I call them "human times". They are, in a way, just conventions to label certain points or ranges in the continuous, physical timeline, but as you can guess, the topic is not as simple and nice. Typical human time labels are already given as examples here and there around this text, so I'm not going to create a separate list of them here.

I would classify human times at least into two different types:

1) The ones that are so widely used and commonly agreed upon that they have evolved into official standards, such as ISO 8601. An example is the Gregorian calendar system that provides us with dates, months, days of week, and when each date starts and ends exactly, depending on the time zone.
2) The ones that vary so much between organizations, situations, and users interpreting them that it has at least so far made little sense to try to unify them into common standards.

Then there are the ones that don't necessarily fall clearly into any of the two categories: I think for example that public holidays in Finland are easily found from official sources, but I'd not rely on standards or software libraries giving me an exhaustive mapping of those holidays to calendar dates, so some human effort and checking would be needed in this case.

The "standard" human times are widely built into software libraries and methods, so most of the time, we can easily derive them using date and/or time values, and a suitable method. For instance, a date and a

"give me the ISO day of week" function together tells me easily the day of week of a date, in the system that is used in Finland (week starts from Monday, days are numbered starting from 1).

The second category, in turn, is more of a messy world that requires human effort. Special days should be defined and stored somewhere, a developer should know what exactly "rush hour" means in terms of clock times during a day, and so on. I think this is a critical area where teams, organizations, and entire domains such as collective transport planning and management have developed a lot of knowledge that often tends to live in human heads - nice, but risky if we wish to onboard new experts easily. The more we could develop common understanding in format of written documentation, the better. On the other hand, not everything can be standardized and made into unambiguous, shared definitions, as long as we depend on human communication and interpretation.

I tend to consider human times as "labels" that are assigned to certain intervals on the continuous (UTC) timeline. For instance, "tomorrow" said on 10.12.2021 in Finland refers to UTC interval `[2021-10-12T21:00:00Z, 2021-11-12T21:00:00Z)` (at integer second precision); and "months of 2021" similarly refer to *multiple / recurring* exact datetime intervals. In this integration between human and "absolute" times, it is worth noting that intervals of certain *duration* do not always transfer nicely to human times. For instance, months can be 28 to 31 days long - so, to build full calendar months on top of individual dates, adding a constant of 30 days is not enough!

Another note is that these human times and their conversion into exact time intervals can be difficult in comparison situations, where we want to set things to a shared scale. A typical example is when we want to visualize some trend over days between two or more years: in that case, it makes little sense to handle dates through weeks, days of week, and days of month. Instead, it is better to use the ordinal day number within a year, so we get each day to a scale from 1 to 366, and the trends during each year can be nicely compared. In contexts other than this, day order number during the year is less intuitive for humans.

Tips for data modelling

- Think carefully before adding human time label fields to your data, like `monday`, `is_weekend`, or `is_morning_rush_7_9`. Can these be derived directly from your original date, time or datetime values? If yes, why not preserve the original values and render any human time labels on the fly?
- Time-related methods and libraries in various programming languages and environments provide excellent ways for extracting necessary parts from dates, times, and datetimes on the fly. Get to know PostgreSQL `EXTRACT` function: `extract(hour FROM tscolumn)` to get hour of day, `extract(isodow FROM date)` to get day of week with `1 = Monday`, and so on.
- For non-standard human time labels, such as your custom "periods of day" like `morning peak, midday, afternoon peak`, think about creating custom functions or lookup tables that help you rendering those labels for date/time/datetime values. Avoid repeated, error-prone use of complicated formulae. Often, management of these special labels is built as part of a transit data model: an example is the special day type management in HSL Jore3, where a lookup table, administered by a human, is needed to match day type labels for exceptional service patterns to exact calendar dates.

1.5 More and more time dimensions...

So far, I've discussed different ways to express time in the "business context" of transit data, in other words, how real-life time concepts are converted into data model features. However, I think that almost any live in at least two different *time horizons*, or *dimensions*:

1) Business time, e.g. "this trip has a planned initial departure date and clock time";

2) Metadata time, e.g. “this trip entry for the next summer was created a month ago”, or “the trip departure time was corrected today”.

If we want, we can split these two horizons into further subclasses. For instance, business time could include *planned* and correspondingly *realized* times, which can be *event times* (happening at a certain instant, or between two instants), or especially for planned features, *validity times* (corresponding to features that seem static but actually change over time, such as stop locations). Metadata time could include time types like feature *creation*, *modification*, *deletion*, or in case of capturing real-time data, *recording* times. I understand *creation* and *recording* a bit differently, since the first could refer more to a human interacting with a data system and creating features, while the latter comes from automation as the system stores real-time data (with some latency).

Below is an example of multiple time dimensions in the same context. Let’s assume that I did the following as a planner:

- On 9.12.2021 (*planning process time*), I finalized a vehicle schedule for route 23 that will be effective between 19.6.-15.8.2022 (*validity time*).
- The route version used there is valid from 1.9.2021 on, with no upper limit (*validity time*), and it uses stops and links that were last modified on 13.8.2021 (*feature modification time*).
- I upload the schedule to our transit registry Jore, and it is published to the operator and to the public in Spring 2021, some months before the summer season. Now the *planned event times*, such as departure and arrival times at stops, are known and expected publicly.
- Then, from 19.6.2022 on, vehicles run the scheduled trips, and we get *realized event times* such as actual departure times from stops, and run times from stop to stop.
- Our AVL system records those realized times as events from the vehicles. Because of inherent latency in the vehicle computers and in the network, the *recording times* of that data will differ slightly from the corresponding realized times.

The exact terms for different time dimensions are kind of improvised - I haven’t learned about good common standard terms for them, or about a classification exactly like this - but I hope the idea is clear: planned and realized times are not the only ones a planner or an analyst should be aware of. There are more dimensions affecting and interacting with each other in data models for both planned data, such as the transit registry Jore3 (and upcoming Jore4), and realized data, such as AVL data warehousing. In my view, all of the time dimensions included in my example above should be taken into account, ideally, but it is of course not quite simple. Imagine a system where *all* the modifications are tracked, and old versions of any modified feature is tracked; imagine that in addition to route versions, every network part such as stops and links are versioned, even using second-precision timestamp ranges, if not date ranges; and imagine what a mess it would then be to take a track of AVL data of a single realized transit trip, and try to join back every planning-related data type to that track, such as contracts, schedules, operation prices, stops, links - and any modifications to these in between - such that the result is consistent and unambiguous.

It would be difficult, indeed - but so is our life with the current data models, where the designers have made simplifications and straight assumptions about planning processes and transit operations reality in rather critical spots. Maybe the most typical example is that in the current Jore3 registry, we have no network versioning: this means that in case of a roadwork, for example, stop locations must be changed and links created, split, and modified *ad-hoc*, and no “old versions” of those network features are saved anywhere. Now, this of course affects how any route version that was effective before those stop and link changes looks like, how long the route geometry is, and so on. Go and try to join positioning data from realized trips to the route geometry representation as it was meant when planning the trips - not going to work. And maybe the worst: the network changes must be done exactly the same day they start in the real life, they cannot be scheduled beforehand. So, simplification on the data modelling side (“only one representation of the network is preserved at each moment”) results in error-prone, laborious, not-so-simple work by humans.

I don’t know the answer, the golden path, how much complexity of a real-life transit system a data model should catch - it depends on the user requirements (that are often really difficult to determine upfront,

though) and resources available for both development and maintenance. However, I would encourage dev teams to embrace the need for multiple time dimensions, coming from different contexts such as validity of a planned object or technical modification times, instead of starting to oversimplify things the very second the reality looks like hard and uncomfortable to model.

We don't have to reinvent the wheel of time and time range modelling for every database table: instead, it'd be better for a developer to create a general mental model of how time instants, durations, validity times, their overlaps or gaps, and other substantial time-related concepts can be implemented in any data model, and in a given technology, such as PostgreSQL. Being familiar with such concepts and tools that enable implementing them easily, a developer is much more prepared to confront the complex reality of things happening on a timeline. Or, I'd say that the timeline as a physical phenomenon is anything but complex, but the complexity and difficulty comes from all the meanings and conventions that humans have assigned to the timeline. This all requires, of course, a lot of practical experience, trial and error over years, and on the other hand, learning about tech and theory. I've already spent many years trying out dramatically failing data models, also over-engineered and needlessly complex ones, and countless hours in documentation pages such as the one about PostgreSQL time and date functions. Now I feel like I've gathered a bit of information on how and how not to do things with time in data.

Tips for data modelling

- There are multiple design patterns to apply to feature versioning over time. One could preserve the latest feature (e.g., row in a database table), and upon any modification, discard the old row or archive it to a different location; or one could build the versioning into the data table using timestamps or timestamp ranges, for example.
- In most cases, I would start from the latter example: in the context of a relational database, design your table as usual - decide a suitable primary key, or multiple ones. E.g., integer `stop_id` for transit `stop` table. Then, add the time versioning layer: a date, timestamp, date range, or timestamp range column. Add that as part of the primary key. If using the range type, think about using an exclusion constraint that prevents range overlaps between rows, in addition to row unicity over the primary key column(s).
- If using date, make sure you know whether you want to refer to real dates or operating days in transit context. If using timestamps, make sure you know whether you want to refer to local or UTC time. (UTC time is often a wise choice in metadata time).
- If the time versioning of the features does not need to support *gaps between versions*, then simple timestamp or date should be enough: successive timestamped versions of, say, a `stop_id` then form a continuous history. But if this history needs to support gaps, e.g., a stop is not available on a certain time range, then date or timestamp range can be better, since you can cap the validity of a version to end before the version validity starts.
- Often, separate date or timestamp columns such as `starts_from_utc` and `ends_to_utc` for validity ranges are preferred to range types. I like single-column time range types, though, since they allow flexible use of range-specific operators. Lower and upper values of ranges can always be exposed to a database user through a view.

1.6 Don't forget the time zone!

This is something I already mentioned about above, but let's discuss it as a final topic: we always have to take into account that in Finland, we're not living in the nice and universal UTC time but have to deal with +2 or +3 hours offset to the UTC time, and that changes according to daylight saving time (DST) changes every March and October. Considering time zones in programming is pretty hard. I think there are hardly any shortcuts in this topic.

It is often said that timestamp / datetime values should be stored in UTC time (e.g., as UNIX decimal

seconds, internally) whenever possible. This generally makes sense, since with UTC time, we can be sure to have no ambiguity due to daylight saving time (DST). However, UTC kind of represents the "physical truth" of continuous time, and is great at pointing at exact instants or ranges in time; but humans use more complicated interpretations and derivatives of time, as we have seen above.

For instance, the calendar dates, operating days, and clock times are in the transit context are always bound to the local time. If we were to force Finnish clock times to UTC times, it would cause shifts back to the previous date, and without the date part attached (remember, date is not there yet with vehicle schedules specific to day types, for instance), we'd lose information.

Luckily, modern programming languages and libraries have advanced methods to handle time zone information and keep it attached to the time values themselves all the way from, say, raw data to a localized UI. But, where this is not possible, such as with `date` data types often, the programmer must be well aware of the time zone that we're handling each time, or wish to handle. I think generous use of words *UTC* and *local* together with date and time variable names and terms in documentation would not be a bad habit.

Tips for data modelling

- Instead of manually converting to UTC time, store absolute datetime values *with* the timezone part - this could be the offset such as 'GMT+02', or standard name such as 'Europe/Helsinki', depending on the system. In PostgreSQL, this means using data type `TIMESTAMPTZ / TIMESTAMP WITH TIME ZONE`.
- You can then render datetimes stored like above in any timezone you want. In PostgreSQL, you can ensure a certain timezone like this: `"tstz col name" AT TIME ZONE 'Europe/Helsinki'`.
- Be aware that client locale settings - in a web browser, in `psql` Postgres prompt, etc. - often affect in which timezone the timestamps are rendered! If you don't explicitly state the timezone like above, you may get different results on your local machine and on a virtual server that lives in UTC time, when querying the date part from a datetime, for instance: `extract(date FROM timestampcolumn)` in PostgreSQL.
- Often you cannot store the time zone information along with a time type. With PostgreSQL data types useful for transit times, this applies to `DATE` and `INTERVAL`, for instance. In these cases, indicate that local time is meant, by naming the objects like `operating_date_local`, or stating this clearly in the documentation.

1.7 Conclusion

Above, I've gone through a number of aspects that can be used to describe and examine the temporal dimensions of collective transport planning and operations:

- Durations, and how to connect times and durations
- Real days and operating days
- Real and 30-hour clock times
- Absolute timestamps (or datetimes)
- Time ranges
- Intervals and "civil" general intervals such as "morning peak"
- Deviations (or time differences)
- Metadata times and temporal feature versioning
- Time zones

This all seems like, and to some extent is, a mess. Let me cite a great blog post about handling time zones in Python: *"Working with human times correctly is complicated, unintuitive, and needs a lot of careful*

attention to detail to get right". Simplifying the concepts of time and in transit too much, and underrating their significance, can lead to concept and data models that cannot catch the complexity of transit, though. In the end, time is *technically* a pretty simple concept: everything happens on a continuous timeline, and humans tend to assign certain meanings to points or ranges on that line, such as days of week, morning rush hours, or start of a year. In many cases, using known reference points in time + doing additions, subtractions and other maths with decimal seconds works quite nicely, and the rest is just visualizing those seconds values in an understandable manner. Just be aware of the context - whether you are dealing with datetimes, dates, clock times, durations, intervals, deviations, or with some other weird aspect to time. And whether you are dealing with "human time" or time as a continuous physical phenomenon. I know it's easier said than done. Anyway, in my view at least, time is a fascinating part of transit data modelling in its simplicity and complexity at the same time!

Reading tips:

- PostgreSQL date/time types
- R `{lubridate}` library for dealing with date and time types - this library provides excellent methods and concepts for time aspects, with a partly similar classification I have applied here. However, `{lubridate}` does not natively understand 30-hour clock times of the transit business, for instance, so you still have to apply your custom chunks of code there…
- GTFS Reference is a great place to learn how dates and times can be handled in planned transit and passenger information context
- Try out the Digitransit GraphQL API to learn how timestamps, durations, and deviations between planned and real times are handled behind HSL Route planner, for example. You'll find things such as wrangling with "integer seconds since midnight".

2 Place in transit data

Words like *place*, *location*, and *position* are commonly used in transit planning and transit data modelling. Just like *time*, these words are overloaded. I have noted that it's often not that clear what exactly is meant: instead, it must be figured out implicitly from the context, which is not always easy, especially for new people, such as software developers, who are only underway onboarding the business domain. We could do better documenting our assumptions and mental models of these terms and their usage.

In this text, I'll try to open up my thoughts about the meanings of place, location, and position when it comes to *transit vehicles*. There's of course much more to the topic than this - for instance, meanings of place for *passengers* moving in the transit system would be a whole different story, but I'll not go into that this time.

About the three terms: I know there are different personal connotations, as well as more or less different official meanings, to *place*, *location*, and *position*. Unfortunately at HSL, at least, we don't have exhaustive definitions for these words to use in data models, for instance, and I think in practice we have a lot of overlap in their usage. I'll not try to give the right answer to the definition question in this post. Rather, I'll be using any one of these for approximately same meanings. Perhaps one nuance to note is that I consider *place* a more general, area-like term, while *location* and especially *position* refer to a more or less exact point in a space. You may well agree or disagree with me!

2.1 Stops - anchor points for vehicles (and passengers)

I'll first discuss transit stops, because they are such an essential part in defining *where* a vehicle is located in a transit system.

A *stop* is a basic unit in a transit system in many ways. It is the *service interface* for passengers who use stops to enter and exit transit vehicles, to wait for them, and to transfer between vehicles. Conversely, for the transit vehicles, stops are usually the only allowed areas to open and close doors, and serve passengers. Therefore, stops comprise the important network of places that any transit system is based on. Strictly thought, the space in between stops is not accessible to passengers (as long as they are travelling by transit and not by walking, for instance).

Stops can be grouped together into more general constructs: most often, stations and terminals are technically collections of stops. GTFS specification uses the `parent_station` attribute to enable this, for example. It may be relevant as well to add grouping labels to stops on opposite sides of a road: technically, such stops are totally separate points of service, they almost never even serve the same route/direction variants, but from a passenger's point of view, they are considered very much belonging to the "same area". The same applies even more to multiple stops on different sides of an intersection, which are often used for transfers.

In schedule run time planning, described in more detail in Chapter 1, not all stops belonging to a route or to a section shared by more routes are relevant, not even nearly. As long as there are several successive stops on a route in a similar traffic infrastructure and having similar passenger demand patterns, the real stop-to-stop vehicle run times tend to behave in a quite linear manner, i.e., the mean speed remains almost constant over multiple stop intervals. Thus, we need to define separate run times only when the mean speed and the conditions defining it change significantly, or whenever we want to define a holding regime (regulating stop) to a stop, to prevent vehicles from departing early. The schedule planning software used at HSL, HASTUS by GIRO Inc., uses the term stop `place` to label stops for these purposes.

In practice at HSL, certain stops get place labels upon need as global attributes, and to actually use a stop as a scheduling or regulating place on a route, the place needs to be *activated* for that route (+ direction and variant). This is because stops may have different roles in different routes, for instance due to different usage patterns by passengers, or because a stop is used for regulating on route A but not on route B. Even more importantly, places are yet another way to group together multiple stops, and often this means even adjacent stops in the same direction on the network; to use places reasonably on a route, only one stop of each place must be chosen for each route.

2.1.1 Are stops points, areas, or linear features?

Stops are most often understood and modelled as points, which makes a lot of sense. Usually the point on a map shows where the stop pole, and possibly shelter, is located, and this point is then easy to navigate to and from, both to passengers and vehicle drivers. Thus I'd say that a point geometry is a bare minimum a stop must have in a transit data model, in addition to a unique stop id.

In reality however, we see that stops require some *area* to work - vehicles cannot stop and passengers cannot wait at a singular geographical point. Practically, vehicles serving passengers always reserve a certain length of the street, or possibly a stop pocket on the side. To match these vehicles to the stop point they are currently serving, we need to model this pocket, or a piece of street, in our data model. Some error margin should also be included, to address random jitter that is inherently there with vehicle GPS signal. An answer to this is *stop radius* that defines a reasonable area, 20 or 30 meters for example, around the stop point geometry. The resulting circle is then used to match any vehicles inside it to the stop. The circle alone is not enough, though: to not match vehicles that happen to traverse through a stop they are not meant to serve according to the scheduled route, the list of route stops (stop pattern) must be involved in the matching algorithm.

A common problem with stop circles is that since they have to be defined large enough to catch some outliers too, due to GPS jitter, they might grow *so* large that they start catching vehicle GPS position that they should not. For instance, a route geometry may often do a "stitch" - visit a dead-end street and then traverse it back to another direction - and such a case often results in vehicle-to-stop matching done too early or late. Another source of error are zigzag-like, self-intersecting route geometries in terminal areas where the same stop radius may again be traversed multiple times by the route, and at the same time, especially in inside terminals the stop radius may be extremely large due to unusually large GPS jitter.

Figure 5 shows an example of a tram stop in Fredrikinkatu/Bulevardi streets in Helsinki, where we have a tram stop modelled in the pole position, but the actual GPS locations of stopped trams (here, from one week) are significantly offset from that point. Blue dots have had the correct `stop_id` matched, while the others didn't have the stop recognized. We can clearly see where the stop radius goes, 35 m around the stop point in this case. The example area has significant GPS jitter probably due to high houses and a narrow street.

To address the aforementioned accuracy problems, stops could be modelled as arbitrary polygons that reflect the real areas actually taken by vehicles - or at least, how GPS sees that area. While this requires some human effort, it could reduce the amount of matching errors. Another alternative could be to always match vehicle positions to the street link they are currently using, and mark the stop *coverage area* as a length interval on the link, or on several successive links. It would then be the responsibility of the GPS-to-street matching algorithm to handle outliers correctly - include or leave them out depending on the route geometry that the vehicle should be following, for example - and the remaining matched points on the links would be a pretty reliable source for stop matching.

A downside of these more precise methods, arbitrary polygons or linear distance coverage intervals, is that they are harder to calculate in realtime environments, and probably harder to debug as well. A stop radius provides a really straightforward basis for calculations in 2D coordinates - it's just about whether the Euclidean distance between the stop and vehicle position falls under a given limit.

Ideally, all of these methods could be used in a data model, each fitting best a certain purpose: stop pole/shelter point would work for passenger information, there might even be another point representation for median *stopping* location, then a radius around it to use quickly in realtime matching, and finally, a polygon and / or a linear distance coverage interval to use for more precise and reliable data processing *afterwards*, when calculating KPIs based on realized stop arrival and departure times, for example.

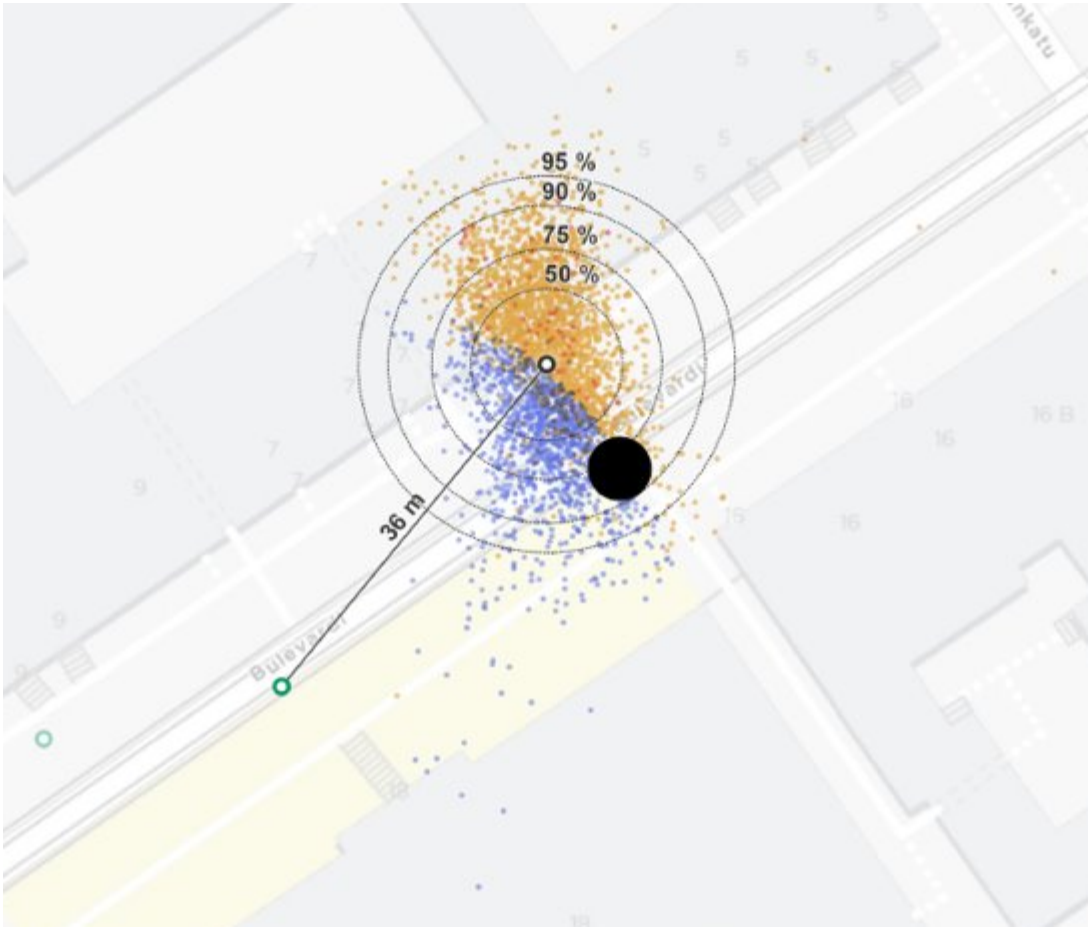

Figure 5: Example of mismatch between modelled and actual stopping locations in Fredrikinkatu, Helsinki.

- Be consistent and careful in how you determine the *point location* of a stop in your data model. Is it to always represent where the stop sign pole is located? Or does it rather represent the median or average position where vehicles actually seem to stop? These can differ even tens of meters for large, busy stops.
- For the aforementioned reason, do not rely on modelling your stops as points only. At least a radius around the point is needed to catch the vehicles that stop near the point at variable positions.
- A stop radius is straightforward to use in vehicle-to-stop matching algorithms, especially in realtime applications. However, it can be too rough for an estimate of the real area used by vehicles at that stop, especially if route geometries traverse through the resulting circle in complicated ways (common in terminal areas, especially), or if successive stops on a route end up with intersecting circles.
- To address that problem, you can model the stop area with a polygon that represents the real stopping area while leaving out "outliers", or by defining the stop as a distance coverage interval along the related road or track links, which then catches correct distance values of vehicle positions projected to the links from GPS coordinates. See Part 2.3 for more information.

2.1.2 When is a vehicle *at stop*?

Stop events are realtime or historical data about actual departure and arrival times of trips by stop on a realized transit trip. They can have other stop-related attributes as well, such as door opening and closing times, and number of boarding and alighting passengers. An ordered list of stop events forms the spatiotemporal position history of a transit vehicle at the accuracy of stops. This is typically an essential source for not only transit planning processes such as run time planning, but also for various key performance indicators (KPI). In my experience, stop events are the most common type of data about realized transit operations in research papers, too.

While stop events are perhaps the most crucial type of realtime data for passenger information and history data for transit analysis, KPIs, and planning, I'd say they are not "real" events. They are artificial in the sense that they depend completely on other, multiple underlying facts: where the stop point was located in the planning database, how the effective stop area was modeled (e.g. circle or polygon around the stop), what the role of the stop was on the trip route and if there was a risk of mismatching e.g. because the route goes through the stop area twice or more; and on the other hand, the accuracy of vehicle positioning (GPS) affects the resulting stop events, too. Since stop events are so prone to various errors reducing their temporal accuracy or even preventing the events to be triggered at all, I would not recommend recording them as the only type of data about realized transit operations. Persisting original vehicle GPS tracks, their trip status history (which scheduled trip they were signed in to at each time), as well as route geometry, network link, and stop representation from the corresponding operating days forms a more reliable basis to not only calculate the stop events but also to monitor their quality, debug, and re-calculate them upon need.

Practical examples of reflecting transit vehicle location through stops are real-time arrival predictions in Digitransit (behind the HSL Route planner, for example), StopTimeEvent message type in GTFS Realtime specification, and stop-related event types in Digitransit HFP (high-frequency positioning).

- If using plain stop events in your data model without a view to how these events were generated, treat them critically. Can you trust that the stop points and areas were created realistically so the vehicles could match the stop easily? Can you trust the accuracy of vehicle computer GPS devices so they have produced precise position values when matching the stop?
- Consider if it's possible to generate, or at least validate, the stop events afterwards with the facts needed for stop matching: pure vehicle position (GPS) data, the original stop locations and areas (such as radius around the stop point), planned trip route geometry, and the stop as part of the route. This way you can verify whether the stop matching process has worked correctly - this can be as simple a process as checking samples of GPS data together with the stops on a map.
- Using those facts rather than stop events generated on the fly in real time, you can adjust the stop location, area, and matching parameters, and re-run the matching process afterwards, if required. Having only stop events matched in real time can be pretty unreliable, since you cannot back up and fix errors like this.
- Storing all these facts and generating stop events afterwards doesn't mean you have to store huge amounts of data. You can define a shorter lifespan for big data, such as raw GPS points: keep them as long as you need for the matching and validation process - say, a week - and discard old data regularly from your database.
- The more critical realistically functioning stop events are for your transit KPIs, the better you should log and monitor their data quality and reliability of their generation process.

2.2 What is between stops

Transit vehicles do not jump from stop to stop, they have to find their way there through a decent path on the transport network. From passenger's perspective, a vehicle can be "stopped at" a stop and thus available to enter and exit from, or not available: "incoming at" (just about to arrive) or "in transit to" (underway) to the next stop. This is how GTFS Realtime sees it. Meanwhile, there's a whole lot going on to the vehicle driver: the exact itinerary to follow in the street network, and various reasons to slow down, stop, and accelerate again, whether it be due to other movers in the same space, traffic lights, or weather conditions. A transit planner or analyst should interested in these between-stops factors as well, since they all contribute to the run times between stops that we wish to minimize for fast service. At least, we want to minimize the *variability* - unreliability, randomness - of them.

In this regard, it is important to identify routes that vehicles follow not only in terms of ordered stops, but also in terms of an exact path of street or track link geometries in the network. Ideally, the path tells about not only the geographical characteristics but also what features and circumstances are involved: traffic lights, pedestrian crossings, bus lanes, speed limits, mean volumes of other traffic (as a proxy of mean "free" speeds in practice). And, of course, stops located along the route path. These have to be considered in two ways: firstly, stops that belong to the route and are therefore a "desired" reason for stopping, and stops as infrastructure that do not necessarily serve that very route but might cause excess stopping and waiting, e.g. due to another transit vehicle in front.

Network paths between a given pair of stops are not set in stone. Transit network are in constant change, more or less, due to infrastructure development, construction sites, temporary disruptions, and so on. For these practical reasons, actual transit vehicle *itineraries* need to be updated every now and then, even if the stops and their order do not change. Moreover, paths between stops are sometimes technically not the shortest paths in the first place. If they were, we could just derive the itineraries automatically from the ordered stop list, assuming we have a decent network model at hand. But this is not the case. Therefore, the stop list and network itinerary of a transit route must be maintained separately in a data model. Of course, stops and route links have common conditions that must be enforced in the data model: the stops of a route

must be available at the links as well that the route uses, and those stops and links must be traversed in sync.

I wish the network aspect could be better considered in common transit data models, such as GTFS. There, changes in stop lists are pretty easy to detect between feed versions, while changes in the underlying network itineraries must be figured out implicitly from geometric differences between route shapes, or from different `shape_dist_traveled` values of the same pair of adjacent stops in `stop_times.txt`. Also in HSL Jore transit registry, network links are there under the hood, but export files and (currently almost non-existent) interfaces provide only entire route geometries and stops belonging to the route. Accessing the ordered link geometries and ids of a route would enable matching vehicles to *link* rather than whole *route* geometries, and eventually it would be easy to see which vehicles are using the same street or track link, no matter which route they are on.

- A transit route pattern can be expressed as an ordered list of stops or as an ordered list of directed street or track links. Ideally, include both of them in your data model, and make sure to validate that these two go hand in hand without logical conflicts.
- Stop and link based paths do not determine each other automatically but require human intervention. A route through a link does not necessarily use all available stops at that link. A route from stop A to stop B does not necessarily use the shortest path of links between the stops.
- Vehicle status in relation to route pattern stops, such as VehicleStopStatus in GTFS-RT, is not a fact coming out of the blue but requires a reliable algorithm to detect whether the vehicle is approaching, stopped at, or departing from a given stop. It'd be nice if your data model not only makes available not only this status information but also how it is generated - e.g., the raw GPS vehicle position, modelled stop locations and areas at that time, as well as parameters used for the matching.

2.3 Vehicles on a map or en route?

As you might have seen above, I generally assume in transit data modelling that locating a real vehicle starts from its GPS position: latitude and longitude coordinates in the WGS84 coordinate system, from which we can convert to any other system, such as metric ETRS-TM35 coordinates that are useful in Finland. This is the "purest" form of vehicle location available to us (in HSL context also publicly available in real time, through HFP), as accurate as the vehicle GPS device can provide. I recommend recording this "pure" data for analysis, KPIs, and planning purposes, not only because it enables examining vehicle movements and GPS accuracy generally in a 2D space, irrespective of the transit network (and its possible modelling errors), but also because we can then re-calculate any derivatives such as *projected* vehicle locations and stop events at any time from raw data, if we note any errors in the calculation parameters or reference data such as stop points or network link geometries.

By *projected* vehicle location I mean the *linear distance value* of a vehicle in relation to a network link or the whole route line geometry that the vehicle is currently traversing. This kind of data is needed for use cases like in Figure 6, where the route planner app needs the vehicle position not only as 2D coordinates for map rendering, but also as a numeric value that expresses its location between stops. Similarly, it can be extremely relevent for certain KPIs and transit performance analysis in general to get vehicle movements matched to corresponding line geometries, such as links, street segments between intersections, or routes, so these can be used as the common 1D dimension for analysing vehicle movements in space and time. My entire Master's thesis project was about this matching thing.

Point-to-line *linear referencing*, available in PostGIS, for instance is an extremely useful tool for producing projected vehicle locations. However, only the point and line geometries are often not enough, since used alone, they can lead to incorrect matches, e.g., when a route geometry intersects itself in a loop. The

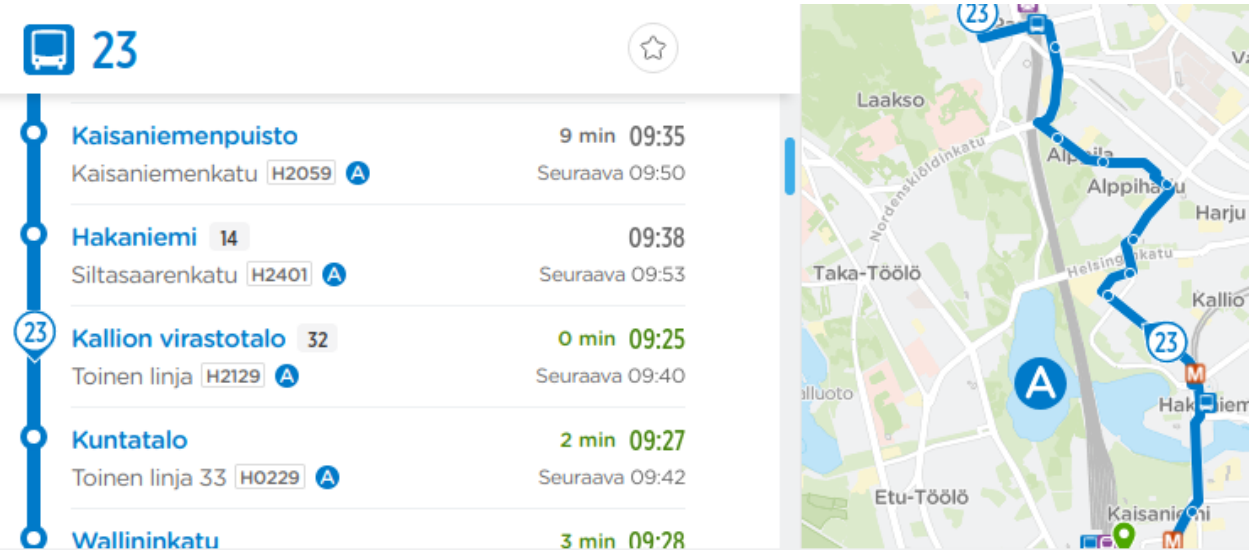

Figure 6: HSL Route planner showing realtime location of a vehicle along the stop list (1D, left) and on a map (2D, right).

matching algorithm should be able to consider other relevant facts too, such as the movement heading of vehicle position points, their speed, and a kind of a lookbehind to previous matched points to determine whether it is more realistic to match a link 5 meters away that would result in sudden jump of 250 meters forwards measured along the route itinerary, or a link 18 meters away that results in a more reasonable, 10 meters' advance measured along the itinerary, with respect to the previous projected point. These are not always easy and intuitive to implement and debug.

2.4 Conclusion

I would say there are at least three essential ways to express *where a transit vehicle serving a trip is located*:

1) Plain GPS coordinates of the vehicle in 2D space.
2) In relation to road or track *network*: "on link X, N meters from the link start". The distance could also be relative, i.e., a proportion from 0 to 1 of the link length.
3) In relation to *stops*: "at stop X" or "between stops X and Y", possibly with a numeric value telling how much distance is left before stop Y, for instance.

Ideally, we could monitor all of these in real time, as well as store them as history data. This way, different purposes such as passenger information, transit KPIs, and planning processes based on history data, requiring vehicle position information in different formats, could all be satisfied. I think it's important to note different needs in realtime and data warehousing contexts: in real time, all of the three ways to express the vehicle location should be available *easily and quickly* - in fact, the more redundancy, the better. In data warehousing on the other hand, redundancy should be avoided in my view: not only because of inefficient use of storage space and processing capacity, but also because storing these three types of data as plain *results* can lead to ambiguity, conflicts, and invalid data, if we note in the end that they do not fully correspond to each other. Instead, in data warehousing we should identify how each of the three data type is constructed logically from the building blocks - GPS locations, network link and stop geometries, routes as link and stop sequences - and make the three accessible as *views* generated by joins of these building blocks. This way, the data would remain consistent and auditable afterwards.

So, in a way, the three ways to express the vehicle location are just views to the same fact, using different frameworks: 2D map, stops, and network links.

The same logic can be applied to other essential components and actors of a transit system, too: where passengers move in time and space, and whether to inspect this "where" in relation to space, stops, vehicles, street network, and so on. Or how transit route itineraries are modelled - starting from stops or from street network links - and how stops and links are dependent on each other.

Modelling the "wheres" of a transit system requires a hierarchy of components working together from top to bottom and vice versa. Coordinate values, route geometries, stop points, and links are not enough alone, but unambiguously defined relationships between them are needed. Such a model is not easy to create, as we have seen many times in the new transit data registry project Jore 4.0 at HSL. Just try to ingest all the essentials of Transmodel as a developer without a long background in transit domain…

As with the time topic discussed in Chapter 1, I recommend not to simplify things that feel uncomfortably complex, but rather to think carefully through the complexity, and document those thoughts, mental models, related data model design decisions, and ultimately the results in understandable human language and pictures. The hardest things are not the data, code and computing themselves, but rather, getting humans to understand each other what they meant when working on these.

3 Useful data tools

In this text, I'll briefly present my favourite software tools for working with transit and movement data, since 2016.

3.1 Excel - a handy scratchpad

Excel is widely used and known, and in that regard, great for data communication, sharing and interoperability. Today, it's still my Swiss army knife in data wrangling - but only for small datasets that need visual inspection and modifications that are easiest to do interactively in the cell-based UI. Nowadays, implementing a full data wrangling and analysis workflow in Excel would be a red flag for me (while not generally, though - I don't expect everyone to like the same tools and workflows I do). This is mainly because workflows are **not serializable and reproducible** in Excel, even if the data itself was. In a workbook filled with formulae and cross-references between cells and sheets, after a while I'm not really able to tell my line of thought and order of creating the analysis and calculation steps. And, if something is broken, it's often extremely difficult to find where that happens, what comes before it and what it affects afterwards. This is the cost of the nice-feeling interactivity of Excel.

Another reason for me not to prefer Excel as data I/O and analysis tool is that it's designed *too* user-centric: Microsoft tries, understandably, abstract away from the user all the messiness a dataset and the required workflow could possibly have, and make everything appear nice and simple. Unfortunately this tends to do more harm than really simplify the user experience. A famous example is how everything in the raw data or user input that might even slightly resemble a date, or Excel's internal date format (integer where `1` corresponds to `1900-01-01`), is forced into a date. In 99 % of my use cases, this is nothing that I want and just makes me spend another half a minute to convert a column back to `text`, for example, and rewrite the contents. It's for a good reason that this feature has inspired so many memes, like Figure 7.

Compared to non-interactive tools, Excel fails with big datasets (say, over a million rows), and it's not very practical for the user, either, to scroll through a high number of rows and order, filter and modify them by eye - I'd say right after the row number is more than a hundred. Cell value formatting is a great thing when making things for visual outputs - I really like to make simple reports with Excel where I get the correct amount of decimal places, and percentage and euro signs attached to my output in a second. But in actual data handling, it can be rather dangerous if Excel likes to render a csv file with only two decimal places for floats, or convert random integers into dates, or truncate timestamps coarser than original, while an alternative tool like Notepad++ shows what the csv *really* looks like, and how the computer will see the raw strings that csv contains.

Meanwhile, Excel is **great** at creating, editing, and - to an extent, keeping in mind the date mess - opening *tabular* data. Visualizing a data table with columns aligned is much nicer and allows for much faster workflows than trying to mess with text editors. One could assume there are plenty of tools that have the Excel-like tabular view but not the cons of Excel (like knowing better than the users how they want their data rendered and converted without asking…), but unfortunately, I haven't yet found convenient ones. For small csv files, Excel is often my tool number one. Even better, if I need to create a couple of simple calculated columns and I don't want to create an R script for such a quick one-off job, Excel is the fastest choice - just open the csv, add columns with simple formulae, and save the result again as csv. By default, not the formulae but their results are saved.

Csv files and Excel just have some caveats to be aware of. The field delimiter is a usual source of problems. Comma `,` is a default setting at least in the Angloamerican context, but then you need to have quotes around any text fields should they contain commas themselves. In this regard, `;` is a safer choice but less seen as default setting in Excel and elsewhere - it requires some more manual steps in data I/O, usually. Tab-delimited files are a nice thing as well, and tabs work well when copying and pasting data interactively from and to Excel - but tabs can be difficult since they are rendered in a way that is easy to mix up with spaces.

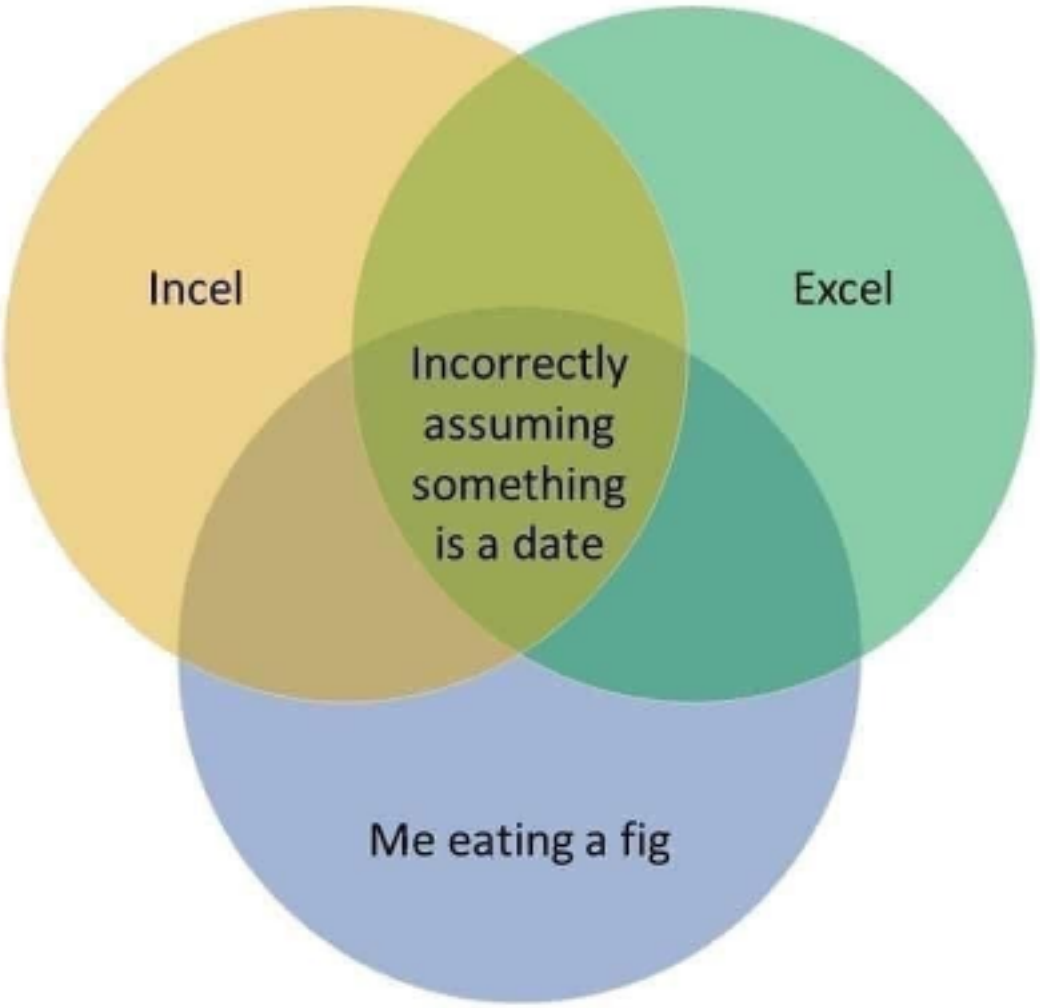

Figure 7: Meme from tumblr.com.

Although I like to criticize Excel a lot, it was originally The Program that taught me how to code and model data - and how not to do it. Starting in 2016 in my first transport modelling job, Excel was my first and only data wrangling tool I used for example when formatting raw data to import into EMME forecasting software. I learned how to deal with field delimiting, certain data types, and formatting data into wide and long formats. Formatting csv or text file data with Excel for another program to read forces one to get rid of empty rows, merged titles, sum cells, many different datasets on the same sheet, and other things that look nice to a human eye but are a pain for machine I/O and data interoperability. Linking sheets and areas through formulae taught me a whole lot about joins and data matching, long before I knew anything about databases and relational data modelling.

For anyone interested in coding at least a bit, I really recommend trying out the next tools, and eventually letting Excel be just part of your workflow, instead of building your whole workflow inside Excel.

Recommended reading:

- Why I migrated from Excel to R by Art Steinmetz
- R for Excel Users, especially Chapter 7: Tidying

3.2 R

R is my great love, so much that I really need to control myself here and keep it short. Briefly put, R is a data-focused programming language and environment that lets you make your data wrangling and analysis steps into a reproducible, trackable process, from which others and, most importantly, future you can learn later. Compared to other programming languages, the best feature of R in my opinion is that it treats any variable as a vector by default, and dealing with vectors and tabular data in R is extremely comfortable. With R, I quickly learned a way of thinking where the programmer does not need to do things tediously element by element, with `for` loops and element-wise `if ... else` conditions, but everything works by *mapping functions to vectors* or other *collections* of values, by default. So, kind of the same as extending a formula into a range of cells in Excel, but done in commands that are located in a logical order and can be re-run at any time. R teaches one quickly to think and transform data in rectangular and tidy format. On top of tidy data, observation-wise calculations as well as pivoting and aggregating data are easy to do.

Anything is possible with R. Anything. There are many things that are absolutely more reasonable to do in other languages and environments, but the multitude of libraries and applications of R surprises me again and again. I think R is at its best in scriptable use cases, where things with data are done step by step in a certain order. There are plenty of applications of R in web servers, interactive dashboards, and other cases where R code is used asynchronously and interactively, though. But for implementing step-wise data manipulation, analysis, and visualization jobs, R is a great tool to familiarize oneself with. At start, forcing things into repeatable steps instead of pointing and clicking in a desktop UI software feels frustrating. In the long run, though, I have noted how R has taught me extremely lot of algorithmic thinking - decomposing problems into computable pieces, seeing which of those individual pieces are in fact essential for solving the problem, and how they can be implemented more efficiently and understandably for humans.

Another nice feature about R is the community and ecosystem around it - people communicating and writing about R tend to be extremely kind, and over the years, the R open source community has become exceptionally helpful and human-centric in many ways, when compared to some other technology niches, I think. R is often used by non-programmers, people who come from "business domains" rather than from computer science, for instance: this often results in their R-focused job to be well understandable by non-technical people as well. Moreover, compared to many other software libraries, R packages tend to be documented in an elegant, comprehensive and human-readable way, and they are often full of practical examples. R itself encourages this through a feature called `vignettes` makes it easy to write and render package feature demonstrations that combine the code and its results in a visual format.

I could keep on telling about how cool R is forever, but I'll now move on the R packages I use and recommend the most:

- tidyverse is a famous R package ecosystem widely used in any data handling use cases with R. It provides elegant and "human-friendly" methods for data wrangling, analysis, and visualization, and has in fact evolved into its own sub-syntax and code smell as opposed to base R. Tidyverse tools are especially useful for functional programming.
- data.table is my choice whenever I want big datasets (over 1M rows) to behave nice and fast even with complex calculations. It has some unique features as well, such as non-equi and range joins.
- sf is my default geographical data tool that works well with spatial data while integrating nicely to tidyverse tools. It uses the so-called Simple Features interface that is also found in PostGIS, for instance. This way, it is easy to implement things in `sf` first locally, and then extend the same logic to a PostGIS server, for instance. lwgeom extends `sf` where the capabilities of its basic features end.
- `DBI` and `RPostgres` are my everyday workhorses for integrating R and PostgreSQL. db.rstudio.com gives maybe the best and most up-to-date overview of working with databases in R.
- Domain-specific packages: `{stplanr}`, `{tidytransit}`

There are also many great transportation domain specific R packages, such as stplanr for sustainable transport planning, and tidytransit for GTFS based transit analysis. While I like their existence and the knowledge ecosystem around them a lot, I have not used them that much as a transit-focused data engineer. These packages provide a lot of nice abstraction and make it easier to focus on the domain problems instead of building the calculation process to solve the problems from scratch - but in fact, I have liked a lot to build things from scratch, and create the solution processes myself. At the end of the day, those packages are using the same core building blocks that I do when doing things with R, and I feel like I would have taken a shortcut and missed a lot of software engineering learning if I had started from those domain packages and not from "pure and basic" R. But, my nature tends to draw me towards the technical issues from the domain issues - someone else would not stand the technical ones a second, which I understand well…

Recommended reading: Geocomputation with R by Robin Lovelace et al.

3.3 Python

Lately, I haven't been using Python that much. I still have a lot of experience in it, originating mostly from Python APIs of transport planning related desktop programs, such as ArcGIS, EMME and QGIS: these are still very useful.

Compared to R, Python is much more widespread and common language, having a strong ecosystem not only in data science but also in web server and even frontend programming, scripting, testing, and what not. Python has great, well-documented, and widely used libraries in data wrangling: an example comparable to `tidyverse` is pandas. And, in geographical data as well: geopandas and Anita Graser's movingpandas for movement data analysis.

In principle, all the things I mentioned with R are possible with Python as well - it's just a matter of taste, and path dependencies. Often people using Python or R, especially the non-programmer ones originally, have started using either one by coincidence. There's no answer to whether R or Python is "better", they can both be used as well and have just some different conventions, ecosystems, and of course, a different syntax and backend. I find it only a nice thing that we have so many good tools and ecosystems to choose from! Why I like R better in many data-related problems is that treating vectors and rectangular data in Python requires some more "boilerplate" code and additional syntax, whereas base R supports those natively, as said above.

I'd strongly recommend Python as the first language to learn, because

- Python is a high level language, with a lot of abstraction over computer internals, making it easy to write, test, and understand;
- but it still has an unbelievable selection of features and tools, making it a good choice in many domains, not only in data science;

- it's so widely used that you'll find an endless amount of good (and also bad) example use cases to learn from;
- most data and GIS related desktop software have Python APIs, which makes them great practical use case environments to test and develop one's new skills;
- code written in Python, a widespread language, is probably better readable and extensible by random other people and organizations than code written in R, for instance.

Recommended reading: Automating GIS-processes

3.4 PostgreSQL

PostgreSQL is an object-relational database management system. It's open source, free to install and use, and my tool number one whenever I want to move away from my local development environment and scale up data storage and transformation, either because of large amount of data to handle (not fitting in RAM anymore), complex datasets (joins, need for indexes for fast access, and so on), or need to make the data available for others than me as well, or all of these reasons.

Although any kinds of objects, such as nested JSON data structures, can be stored and queried in PostgreSQL, it teaches one to model reality to data structures in terms of tidy, rectangular datasets that are easy to join together with minimal ambiguity, redundancy and repetition. R is a great start here, and PostgreSQL (and any other SQL database engine, too) takes these principles even further - the database environment is some less flexible than R, since declaring tables, data types, relationships, indexes, views, and functions requires some more effort and planning there. For this reason, PostgreSQL makes for a good basis for persistent, "conservative" data whose structure and usage patterns are not changed and experimented with often but that on the other hand work quickly, reliably and are easy to scale up. And because of this, I'd say that R, Python and other flexibly typed, interpreted languages are the correct place to start testing, developing and experimenting with data, and one should move on to databases only when the data model has matured and settled enough. This is something that I learned a lot from in my thesis project - the development was slow most of the time, mostly because I tried to implement things in PostgreSQL right from the beginning. And unfortunately, PostgreSQL does not provide very nice syntax and tooling for quick experimenting, trial, and error.

One of the great features of PostgreSQL is the extension ecosystem - extensions are mainly freely available, open source as well, and often pretty easy to install. My favourite extensions to PostgreSQL are

- PostGIS that adds support for GIS data types and functions. PostGIS has a wide scale of features, and some features are not that generally available in other tools and environments. For instance, I use linear referencing a lot - it basically means things like matching points to lines and using their relative distance value along the line, not the X/Y location of the point. Very useful in transit analysis, and often the only reason for me to use PostgreSQL & PostGIS instead of R for a use case.
- pgRouting, in turn, extends PostGIS. It makes use of topology relationships of points and lines, and enables things like routing and network analysis algorithms. Again, very useful in transit analysis domain.
- TimescaleDB adds support for automatic time column based *partitioning* of tables, meaning that a large table (e.g., over 100M rows) can be easily distributed into many tables under the hood, and thus the table can be efficiently queried using many parallel queries. In addition to this, Timescale provides a number of useful time series functions, such as "bucketing" of timestamps to arbitrary precision, such as 15 minutes.

Recommended reading:

- The Art of PostgreSQL by Dimitri Fontaine
- Crunchy Data Blog
- Writings by Paul Ramsey, a PostGIS core contributor

3.5 QGIS

QGIS is my default, and today only, choice for a desktop GIS software that I can use to

1) explore any spatial data on the map, often supporting R and command line tools such as PostgreSQL client `psql` as part of my workflow;
2) combine, transform, and visualize spatial data *ad hoc*;
3) create automated Atlas reports, like "print a map view around all of these 50 stops, unique by `stop_id`, centering the map view by the stop location and choosing the scale from the `stop` attribute table".

Perhaps my favourite feature with QGIS are the expressions, which allow calculating an arbitrary value for almost any attribute available in not only the datasets in the project, but also the QGIS itself. Similarly, expressions can read attribute values from almost anywhere in the program. This way, I can easily make, for instance, layer visualization parameters such as colors, transparencies, sizes, and scales interactively dependent on the feature attribute values and project state, such as the current map view extent.

QGIS is so much more than this only, but I've not really used most of its features that much yet. For instance, many of the things that the Python API could handle elegantly, I have liked to handle in R myself. Another example is relational data modelling and joins that would be available to do to a large extent inside QGIS, in a flexible and nicely responsive way, but so far, I have preferred doing that kind of modelling on the data source side, such as in PostgreSQL, and use QGIS to render the (almost) ready-made results. But this is of course a matter of taste, context, and requirements, again.

Recommended reading: Movement Data in GIS by Anita Graser

3.6 Git

Git is a tool for local as well as distributed version control of software source code and documentation. Lately though, it has become more and more popular in use cases not necessarily related to buildable or executable code - git can be a convenient choice for any content expressed in machine- and human-readable *text*. (Data other than text can be tracked in git as well, but it's not as practical in terms of version history). Basically, git allows tracking the lifespan of files and individual text lines in them, making changes in controllable units (branches, commits and merges), controlling conflicts of overlapping modifications of the same content, and in case of a failure for instance, reverting back to any state of the content in history.

I think git is mostly useful in managing *teamwork* around a project where several people work on same files and tasks. However, I've adopted it in my personal workflow as well, because

1) it's a nice tool for storing the history of my modifications and work, compared to "dump" copies of the project repository every now and then;
2) the git commits and branching force me to think my work and tasks to do in logical, limited pieces, instead of making random modifications ad hoc here and there.

The latter often becomes a nuisance, though, and often I find myself creating awful dump commits with commit messages like "WIP". For experimental workflows, I would not recommend git, since it requires some setup every time, and adhering to a reasonable commit "discipline", to get the benefits of version control.

A nice example of a non-software project managed with git and GitHub is the open source book Geocomputation with R that I already recommended under Chapter **??** but whose GitHub repository is worth checking as well.

Recommended reading: Happy Git and GitHub for the useR by Jenny Bryan & Jim Hester

3.7 Bash and other command line tools

Bash is one of the most used command line interpreters in Linux and other UNIX-based operating systems (including Mac OS X). It's actually just one of the many UNIX command line tools, but in my setup, it's the basic command line environment to start from every time I open a terminal on my computer.

Compared to desktop UIs, bash and other text-based command line tools provide an extremely efficient, scalable, scriptable, serializable and thus reproducible interface to the operating system, files, and their contents. With the command line, I can easily handle tasks like renaming 1000 csv files reliably based on a certain pattern, or adding a header row to every one of them, or importing all of them into PostgreSQL in an individual process while saving the output and any errors or warnings into a log file - everything in a couple of lines of commands *piped* together. So, bash and other tools provide me with a selection of building blocks that I can use creatively, in a modular manner, to automate my workflows that would take ages and be very error prone if done "by hand".

Recommended reading: Data Science at the Command Line by Jeroen Janssens

3.8 Final words

All of the tools I described here, except MS Excel, are open source software - freely available, and open for anybody to contribute to the source code development. Why is open source so cool, in addition to being free of charge to the user?

Often in proprietary software, such as ArcGIS for geographical data tasks or SPSS for statistical modelling, a user pays a high price to get a *reliable*, *usable* and *useful* product to solve business problems, and often the user expects good *support service* from the vendor to use the product effectively. This is very fine and OK in many cases, in my view. However, I personally find that proprietary products fail to teach the user how *things really work*. They tend to provide comfortable solutions, such as a map view with a ready-made routing engine - but then the basemap is locked to a certain vendor, instead of providing a standard interface to use with *any* basemap service of choice, and the routing engine is provided by another single vendor, such as Google, and the user has no choice in changing it to an OpenStreetMap based one for instance. This is an extreme example - the case is not that bad with ArcGIS for instance, luckily. But you may get the point: it is tempting to use nice-looking, simple, ready-made products, but then you give up on a lot of things to learn and apply yourself, which in the long run could be much more beneficial in your data / GIS career than having learnt only how a certain vendor has thought through the domain problems and what specific solutions they have to them. By committing to proprietary products, not only companies are stuck in vendor locks, but also the individuals working with and learning only the vendor-specific solutions and ways of thinking.

Open source tools have traditionally not been as nicely designed and user friendly as the proprietary ones, but a whole lot of development has been done in recent years: QGIS is no more the buggy, weird-looking niche tool liked by development-oriented GIS people, for example. R and Python have taken over more and more space in data science from desktop tools such as SPSS lately as well. I don't really know if this has happened because some critical point has been crossed, after which enough universities, companies and other actors have adopted those open source tools, which in turn has brought in more users applying, benchmarking and for instance blogging about them, as well as more interested developers to make the tools even better; or because the business model around open source software (e.g., support and expert services, instead of monolith product development) has become more sustainable and widespread. It would be interesting to know. But in any case, the role of open source software has become larger, and I'm happy about that, especially because that makes learning in a wide sense much easier and more inspiring: now not only the GIS and data science products themselves, and how to use those products, are available to learn from, but also their internals, the standards, data types, programming languages, and interfaces they are based on. With open source products, the end users can choose whether to use the "default" products, or whether to take just some of their building blocks to develop a tailored solution themselves. For instance, QGIS is extremely easy to start using from scratch, but meanwhile, it can be customized to a large extent

for different purposes, modules from it can be used without the desktop UI part at all, it provides a server module, a Python interface, and so on.

Open source is not a happy and ideal field only, though. It's pretty sick how many software libraries and components extremely essential to the modern web, for instance, and thus also for many proprietary products from which big companies profit a lot of money, are maintained and developed by volunteers who basically get their compensation through charity, not through sustainable and fair business models. Even if the developers get fair monetary compensation for their work, it's still problematic how dependent we are in many cases on their good will and commitment. For instance, the developers of the log4j logging library were originally, and still are as far as I know, a rather small group of experts who created the library voluntarily: and because of a long path dependency over time, the library became widespread in countless Java applications, and now that the famous security vulnerability was found in it, the original, small developer group is under a huge pressure to get fixes done. Who's responsible? Nobody can fairly hold those developers accountable for the vulnerability, since it was not their choice in the beginning that millions of applications ended up using that software component which was freely available.

I still think that open source continues to take over a larger role in the software world, and thus people, including me, are able to learn more and deeper about how software and code can actually help them solve their business problems. Open source has gave me so much in the last years that I really hope I'm soon experienced enough to give something back as well - contribute to source code development, documentation, and tutorials, for instance. I also wish to work for employers in the future who pay for such work: work that is not counted as their direct profit but benefits them through better expertise and knowledge of the one contributing to the OS development, and through the development, eventually better and better products for the business problems.