Data model sketching

Introduction

Intention:
- Enable low-friction collaboration before committing to any SQL schema

Desirable outcomes:
- Clarify infrastructure network data model enough to probably enable importing routes from Jore3
- SQL schema -like structure

Suggestions:
- Work asynchronously by default
- Suggest a synchronous session in Slack if you deem it useful
- Write down your ideas freely, create suggestions for others, comment
- Once the initial schema is committed to in a documented manner, we delete this document

Roles:
- haphut copies this document to the wiki as such before every review session

Content

How are schemas named and bounded?

Start from Transmodel if it makes sense.

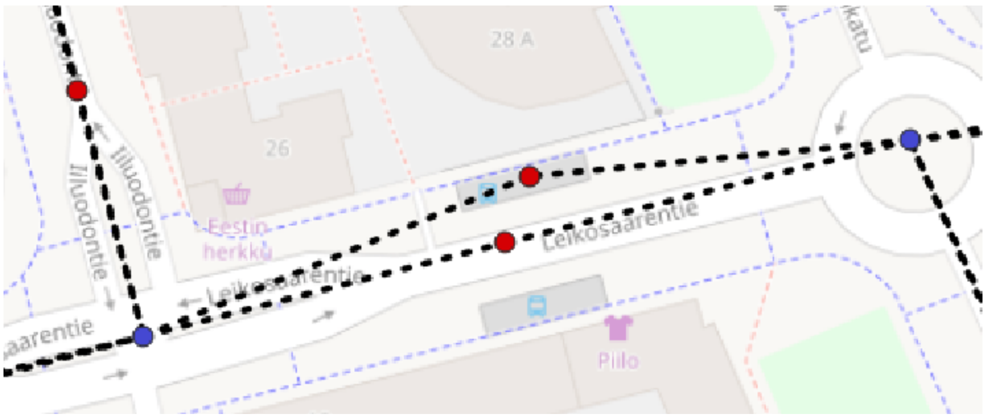
Having a separate schema for the infrastructure network makes sense as HSL does not own or make decisions regarding the infrastructure network. In that sense there's hope that at some point the responsible organizations will offer usable APIs so that HSL would not need its own service for the infrastructure network.

Jore 3

Challenges in the legacy data model

Bus stops are part of the infrastructure layer.

- Moving a bus stop always requires altering the related links (e.g. splitting an existing link), and thus affects routes

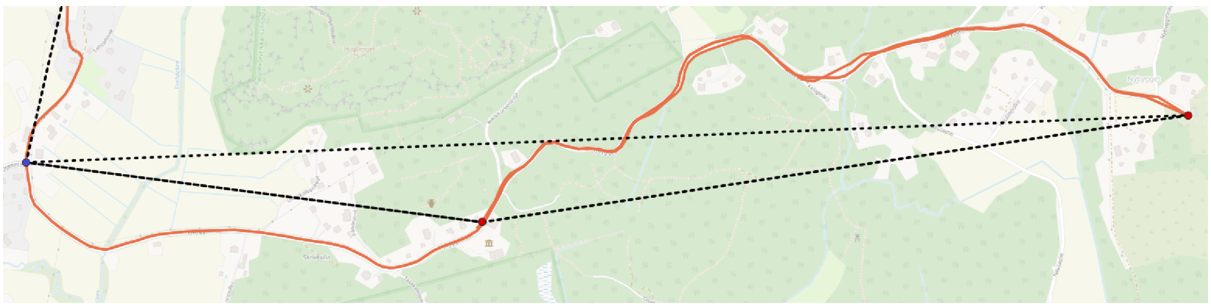


Routes consist of consecutive route links between two nodes

- Each link describes the attributes of the starting node
- As there are 1 fewer links than nodes, an extra link must be added to describe the final node

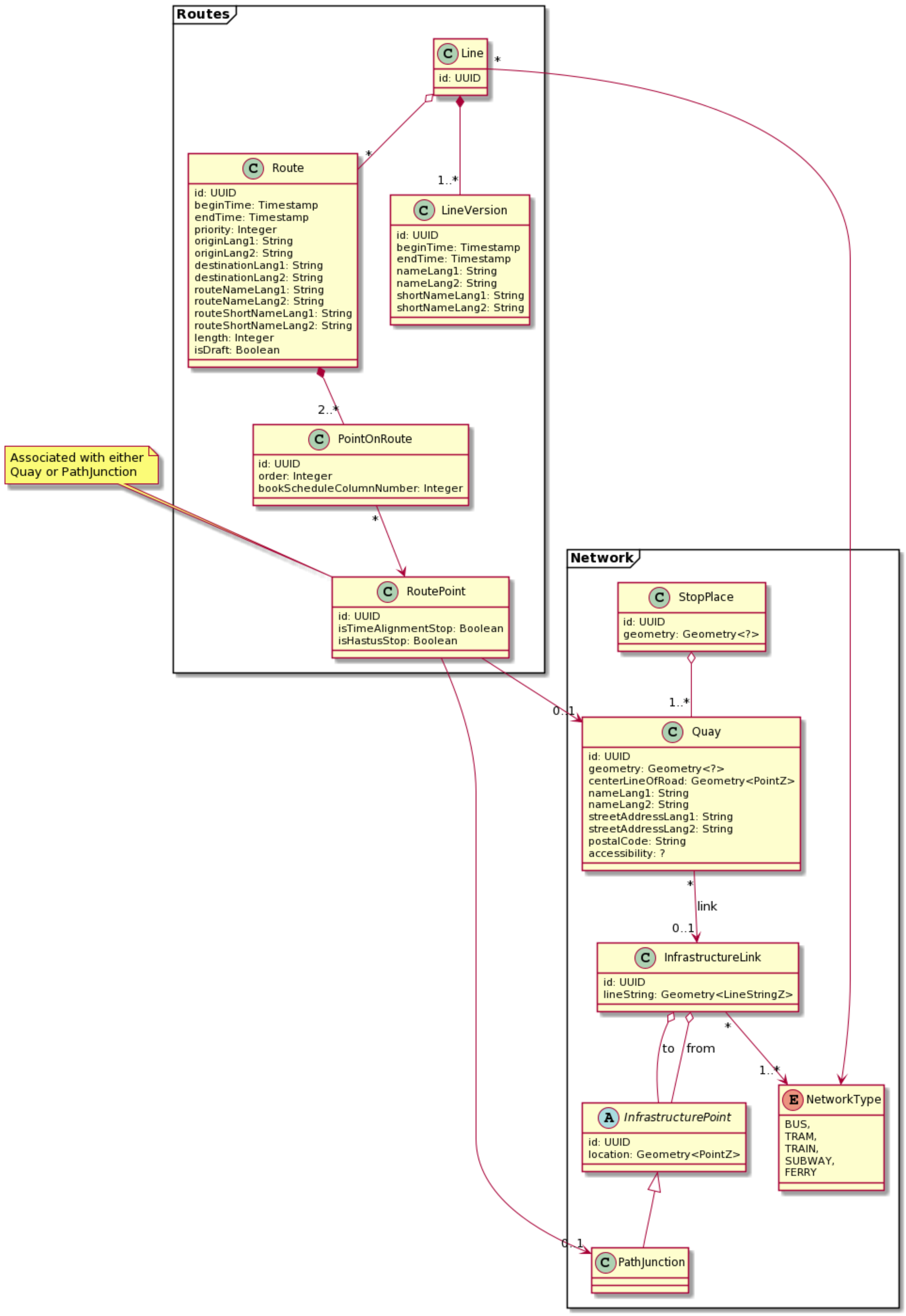
Route links and infrastructure links are the same

- Each link can contain multiple points, which describe the shape of the link. These shapes are drawn by hand using a reference map
- One route may go directly from A->C with a single link while another route might take two links to traverse A->B->C
    - This results in the segment A->C having 2 separate shapes



What HSL has called “reitti” has two aspects: route geometry along the infrastructure network and the stop sequence. As a consequence of having stops as a part of the infrastructure layer in Jore3, the workflows of managing these two aspects are mixed.

Jarkko’s proposal



- Hahmottelin yksittäiset pysäkit siten, että ne viittaavat johonkin tielinkkiin (ovat sen varrella) mutta eivät lukeudu tielinkkien välisiin solmuihin. Näin ollen pysäkin paikkaa siirrettäessä ei tielinkkeihin tarvitse koskea. Kuten Jore3:ssa, pysäkillä on tien keskiviivalle projisoitu koordinaatti.
- Hahmottelin reitit järjestettynä pistelistana reittilinkkien sijaan käyttäen RoutePoint- ja PointOnRoute-käsitteitä, jotka ovat Transmodelista tuttuja. RoutePoint- ja PointOnRoute-käsitteet mahdollistavat sykliset reitit, missä reitti voi kulkea saman solmun kautta useammin kuin kerran.
- Hahmotelmassa RoutePoint viittaa joko pysäkkiin tai katuinfran risteyssolmuun (tielinkin päätepiste).
- Tietomalli vastaa mm. seuraaviin kysymyksiin:
    - Mitä pysäkkejä on tielinkin varrella?
    - Mitä reittejä kulkee tietyn tielinkin tai risteyssolmun tai pysäkin kautta?
    - Jos pysäkin paikkaa muutetaan, mitä reittejä pitää päivittää uusilla versioilla?
- Hahmotelmasta puuttuu paljon olennaisiakin kenttiä, joiden lisääminen saattaa aiheuttaa päänvaivaa, koska malli ei vastaa Jore3:sta (tunnistin jo jotain Avoimia kysymyksiä -kohdassa)
- Jos valitaan geometrioiden koordinaatistojärjestelmäksi sellainen, jonka yksikkö on metrinen (eikä aste kuten EPSG:4326 tapauksessa), se voi tehdä kehittäjän työstä helpompaa joiltakin osin mm. PostGIS-funktioiden kanssa. On esimerkiksi ymmärrettävämpää, jos koodissa kysytään “hae pysäkit 5 metrin säteellä risteyksen X koordinaateista” kuin “hae pysäkit 0,0017 asteen säteellä risteyksen X koordinaateista”. Toki koordinaatistomuunnokset ovat mahdollisia ennen PostGIS-funktioiden kutsumista, mutta koordinaatiston valinnassa lienee hyvä painottaa myös kehittäjäystävällisyyttä.

Open questions

- Do we need a concept for *RoutePathLink* or can we just model a sequence of *PointsOnRoute*?
    - How to model “Via-nimet”
    - A route link might have a different *transit type* than the line. How should this be handled?

RFC

Separating Model Contexts with SQL Views

Premise: specifying the whole data model (tables, columns, indexis..) spanning all modules and integrations is difficult and rigid and may lead to “analysis paralysis”. Changes to the model are difficult, especially if the migrations apply to the whole model and are owned by a single process/entity/repo.

Suggestion: Perhaps we can split the work by defining the boundaries between modules using (read-only?) SQL Views?

- Separate module-specific implementation (tables..) and the API (views)

    - Achieve the traditional benefits of APIs: implementation can be altered as long as the API contract (= the views) is not violated
- Might lower the risk of creating a “big ball of mud” datamodel
- Modules could own their implementation and (if necessary) handle the migrations

Issues and open questions:

- Should we use views or materialized views?
    - There might be some performance benefits with e.g. the importer applications which can rematerialize the view(s) after a batch import is complete.
    - On the other hand working with materialized views might be difficult if we’re using Change-Data-Capture instead of batches.
    - Difference in creation semantics: Materialized views support CREATE IF NOT EXISTS, while views support CREATE OR REPLACE
    - Materialized views support indexes
- You can’t create an index for a (non-materialized) view (directly):
    - The module implementing the view(s) must make sure the actual tables underneath have appropriate indexes
- Do we need a “common” module for shared stuff
    - E.g. a “languages” table (see discussion below wrt. multilingual strings), which other modules can refer to

How to Handle Multilingual Strings

Premise: Some entities (e.g. lines, routes) have one or more fields which must be localized to several languages. The current project needs to support Finnish and Swedish, but the software should perhaps be applicable to other languages and regions too.

Also note that some sources (e.g. Jore 3) might not always provide all possible translations for all entities for all languages.

Option #1 (same as Jore3): Separate columns for each localized string

Example: `name_fi`, `name_sv`, ..
Good:

- Simple to implement
- Version history easy to store (with temporal tables)

Bad:

- Hardcoded localization: Languages shouldn’t be encoded in the database schema
- Hardcoded plurality: We can support only two languages

Option #2: Use JSON(B) to store localizations

Example: `{“fi_FI”: “Nimi”, “sv_SV”: “Namn”, …}`
Good:

- Relatively simple to implement
- Version history easy to store
- Any number of languages

Bad

- No way(?) to ensure on the database level that column values actually match the expected JSON schema
- No way to know what languages are supported

Option #3: Separate tables for localized strings

Example:

- Languages table:
  - language_code text
- Names table:
  - name_id uuid
  - name_language_code text ref languages.language_code
  - name text
- Attribute links table (one per attribute?):
  - link_id uuid
  - entity_id uuid ref target_entity.id
  - name_id uuid ref names.name_id

Good:

- Any number of languages
- Supported languages explicitly listed
- Finding entity by its name (or by other attribute) in any language is easy to implement and does not involve hard-coding column names for different languages

- `language_code` can be used in queries to filter out languages user doesn't care about

Bad

- Separate table for each localized attribute(?)
    - Unclear how many localized attributes there are. 10? 20? Most likely not hundreds.
- Batch inserts/updates might be cumbersome
    - There are at most a few thousand entities with localized strings: shouldn't take too long even if we can't batch everything.
- Unclear how rows should be versioned
- While retrieving rows from an entity table, the names are not shown without joining localization table which makes debugging a bit awkward
- What about tables in different PostgreSQL schemas? Should each schema have a separate localizations table? Or is there a shared one?

Option #4a: Separate codeset and localization tables for attribute names

Similar to option #3. Models localised names for different attributes of entities as codesets. A codeset name could be e.g. "route_short_name". These codesets can exist in source code as enum values in order to make referencing them in SQL queries less error-prone. Differs from option #3 also in primary keys used in referencing attribute name localizations.

Example:

- languages table:
    - language_code text
- codesets table:
    - codeset_name text
- attribute_names table:
    - entity_id uuid ref target_entity.id
    - codeset_name text ref codesets.codeset_name
    - language_code text ref languages.language_code
    - primary key(entity_id, codeset_name, language_code)

Option #4b: Separate codeset and localization tables for attribute names

Very similar to option #4a. Models localised names for different attributes of entities as codesets. A codeset name could be e.g. "route_short_name". The codeset names can exist in source code as enum values in order to make referencing them in SQL queries less error-prone. Differs from option #4a in such a way that codesets table has also an UUID column which makes changing a codeset name easier but involves an additional join in SQL queries.

Example:

- languages table:
    - language_code text
- codesets table:
    - codeset_id uuid

    - codeset_name text
- attribute_names table:
    - entity_id uuid ref target_entity.id
    - codeset_id uuid ref codesets.codeset_id
    - language_code text ref languages.language_code
    - primary key(entity_id, codeset_id, language_code)

How are multilingual strings handled in e.g. Entur/Tiamat?

- Entities have embedded fields (name, short_name, description) with some language (lang)
- Entities are linked to one or more alternative_names, which define some fields (name, qualifier_name, short_name) in a particular language
    - Not clear whether these specify a single field or multiple. E.g. do you need 3 alternative_texts to specify name, short_name and description or 1.
- Separate table (alternative_name_valid_betweens) to keep track what is the valid time for a particular alternative_name row

https://github.com/entur/tiamat/blob/master/src/main/resources/db/migration/V1__Base_version.sql#L352