ECOLE
POLYTECHNIQUE
DE BRUXELLES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES

Large-Scale Visualization of Mobility Data

Master's thesis submitted in order to be awarded
the Master's Degree in Computer Science and Engineering

Ayoub El Hamri

Supervisor
Professor Esteban Zimányi

Department
Computer and Decision Engineering

Academic Year
2025–2026

ECOLE
POLYTECHNIQUE
DE BRUXELLES

DOCUMENT TO BE INCLUDED IN THE MASTER THESIS

CONSULTATION OF THE MASTER THESIS

I, the undersigned

NAME EL HAMRI

FIRST NAME Ayoub

MASTER PROGRAM Master en ingénieur civil en informatique, finalité spécialisée

MASTER THESIS TITLE Large-Scale Visualization of Mobility Data

AUTHORIZE*

~~**REFUSE**~~*

(IN CASE OF REFUSAL, THIS DOCUMENT MUST BE COUNTERSIGNED BY THE MASTER THESIS SUPERVISOR)

(Delete as appropriate)*

Consultation of this master thesis by users of the libraries of the Université libre de Bruxelles.

If consultation is authorised, the undersigned hereby grants the Université libre de Bruxelles a free and non-exclusive licence, for the duration of the legal term of protection of the work, to reproduce and communicate to the public the above-mentioned work, on graphic or electronic media, in order to enable users of the libraries of the ULB and other institutions to consult it within the framework of inter-library loan.

Brussels, 25/05/2026

Signature of the student

Signature of the supervisor
(only if consultation is refused)

Abstract

Author: Ayoub El Hamri
Master's Degree: Master in Computer Science and Engineering
Academic Year: 2025–2026
Thesis Title: Large-Scale Visualization of Mobility Data

Bus and tram operators now publish their fleets' positions live. Turning those feeds into a city-scale map that can be scrolled, zoomed and queried is much harder than the storage problem behind it. The storage side has been solved by MobilityDB (PostgreSQL extension for moving-object types) (MOBDB) [1]; the visualisation side has not. Browsing the same data in QGIS through the existing plugin runs out of memory before the full Brussels Société des Transports Intercommunaux de Bruxelles (STIB) fleet fits.

I close that gap on the live STIB / Maatschappij voor het Intercommunaal Vervoer te Brussel (STIB-MIVB) and De Lijn feeds and report when each of two solutions is the better fit.

(i) A systematic comparison of six rendering pipelines for animated MOBDB trajectories in QGIS, on two contrasting datasets, identifies three deployment tiers. A targeted optimisation of Manzer's [2] published canvas-item architecture reaches ∼181 FPS on STIB and ∼65 FPS on AIS, the only condition to clear the 60 FPS budget on both.

(ii) **RTDataHub** is a three-layer platform. It ingests the STIB-MIVB and De Lijn feeds in real time, normalises both into `tgeompoint`, persists them through a hot/cold loader, and serves them as vector tiles via a Flask Application Programming Interface (API) to a small MapLibre frontend. The platform also runs a bus-bunching alert detector. Median ingestion latency is 10.2 s for STIB and 11.3 s for De Lijn. A seven-day benchmark of the trip-reconstruction matcher places the deployed two-stage matcher (`greedy_chainmerge`) ahead of the vendored MobilityTwin baseline, at a median trip coverage of 54.5%.

(iii) A direct benchmark on identical hardware over nine scales draws the boundary between the desktop and web tracks. The two curves cross at $N_{\text{cross}} \approx 19{,}000$ trips. The desktop pipeline is therefore operationally sufficient at the scales that cover the great majority of analytical use cases; the web stack is necessary at the full live STIB working-set size.

Keywords: Mobility data; MobilityDB; **RTDataHub**; vector tiles; real-time public transport; Brussels.

Acknowledgments

I thank my supervisor, Professor Esteban Zimányi, for trusting me with a topic at the intersection of his research (MobilityDB) and a real operational need at the city scale. His insistence on reproducibility and honest measurement shaped the structure of this manuscript.

I am grateful to the STIB-MIVB and De Lijn Open Data teams for maintaining the real-time feeds that make this work possible, and to the open-source maintainers of MobilityDB, MEOS, MapLibre and PostGIS.

I thank Dawid Banas for the collaborative MEOS.js benchmark of Section 7.13.2; the Node.js reference server and the raw measurements reported there are his work.

I also thank my family and friends for their support throughout the year.

Table of Contents

List of Figures

List of Tables

List of Acronyms

RAM Random Access Memory 1
SSD Solid-State Drive 10
SLA Service Level Agreement 2
SNCB Société Nationale des Chemins de fer Belges 14
SQL Structured Query Language 9
STIB Société des Transports Intercommunaux de Bruxelles 3
STIB-MIVB STIB / Maatschappij voor het Intercommunaal Vervoer te Brussel 3
TEC Transport En Commun (Walloon public transport operator) 14
TTFP Time-to-First-Pixel 24
VM Virtual Machine 10
WASM WebAssembly (portable binary instruction format for browsers) 114
WebGL Web Graphics Library (browser-side 3D/2D GPU API) 9

Notation conventions. Throughout the manuscript, code identifiers and SQL keywords use a `monospace` font. Operator names from MobilityDB (*appendInstant*, *atPeriod*, *minDistance*) are typeset in italic; their formal definitions appear in Section 2.3. The five Brussels-area transit operators visible in the data are abbreviated STIB (= STIB-MIVB, urban network), *De Lijn* (Flemish regional buses serving the Brussels periphery), Transport En Commun (Walloon public transport operator) (TEC) (Walloon, mostly outside the study area), and Nationale Maatschappij der Belgische Spoorwegen (Belgian Railways) (NMBS) / Société Nationale des Chemins de fer Belges (SNCB) (national rail). Unless stated otherwise, *the platform* or RTDataHub denotes the live system contributed by this thesis.

Cross-language note. A handful of concepts have a Belgian French short form: General Data Protection Regulation (GDPR) corresponds to RGPD, Geographic Information System (GIS) to SIG, and NMBS to its French alias SNCB. We use the English form throughout the manuscript and only the French form when quoting Belgian operational documents.

Chapter 1

Introduction

The growing deployment of Global Positioning System (GPS) sensors aboard public transport vehicles generates large volumes of streaming spatiotemporal data, and public-mobility operators want to use this data for live monitoring, post-hoc analysis, and policy evaluation. The technical layer of this data, and continuous *trajectories* of moving objects, is, however, surprisingly hard to handle in interactive visualisation tools. Desktop GIS packages were not designed for trajectories that change at every frame, and lightweight web mapping libraries do not speak the spatiotemporal algebra of trajectory databases (i.e. the typed operators a database exposes for moving objects). Between the database and the human eye, what should be a routine read becomes the project's bottleneck.

This thesis takes a real Belgian operational setting, the live feeds of STIB-MIVB (Brussels urban transport) and De Lijn (Flemish regional buses serving the Brussels periphery), and asks how those feeds can be turned into a continuously running, queryable, scrollable visualisation at city scale. The work is grounded in MOBDB, the PostgreSQL/PostGIS extension co-authored by the supervisor of this thesis, which provides the `tgeompoint` type (a moving point in space and time) and the operator algebra for moving objects. The contributions are software artefacts (a refactored QGIS plugin for visualising MobilityDB trajectories (MOVE) and a from-scratch web platform called RTDataHub), measurements that document when each artefact is the natural choice, and a head-to-head comparison.

1.1 Context and Motivation

Three trends have converged over the past two decades. Geolocation is everywhere (GPS-equipped smartphones, Automatic Vehicle Location (AVL)-instrumented fleets, Automatic Identification System (maritime tracking) (AIS), ADS-B), trajectory storage has matured around the Moving Object Database (MOD) framework [3] and its open-source instance MOBDB [1], and European transit operators publish real-time feeds in GTFS-Realtime (GTFS-RT) [4]. The consumption side has not kept pace: desktop GIS tools like QGIS were not designed for continuous trajectories [5], and web libraries (MapLibre, deck.gl) are disconnected from MOBDB's operators [6].

The present work grounds the gap in Brussels: STIB-MIVB's open real-time feed [7] and De Lijn's GTFS-RT feed [8] together emit on the order of 2 million position updates per day, an almost ideal stress test at metropolitan scale. The work continues Manzer's MOVE plugin for QGIS [2]: the first functional bridge between MOBDB and a desktop GIS, but one whose architecture saturates 32 GiB of Random Access Memory (RAM) before the full STIB corpus can be loaded [2, p. 48]. We treat that ceiling as the starting point.

1.2 Problem Statement

The central problem is:

> *How can the MOBDB ecosystem be coupled with desktop and web rendering stacks so as to deliver expressive, low-latency, interactive trajectory visualisation at the metropolitan scale of STIB-MIVB and De Lijn, while keeping the engineering cost compatible with a public mobility operator's resources?*

Section 3 quantifies the workload, makes the two structural failures of Manzer's architecture explicit [2], and explains why an off-the-shelf web tile pipeline does not, by itself, close the gap.

1.3 Objectives and Research Questions

The general objective is to deliver, demonstrate and characterise two complementary visualisation tracks for trajectory data, one inside QGIS, one in a browser, that share a single MOBDB backend. The objective decomposes into three research questions, each formulated as an answerable engineering question with a measurable outcome:

RQ1 — Cross-architecture animation pipelines for QGIS. *For the animated visualisation of MOBDB trajectories in QGIS at the city-scale workload of this thesis (∼5 000 simultaneous trips), which pipeline architecture among the practically deployable alternatives offers the best latency-versus-memory trade-off, and how does this trade-off depend on the geometric complexity of the underlying dataset?*
Hypothesis: two predictions follow from the question's two halves. First, no single pipeline dominates on both latency and memory — the practical answer is not one universal winner, but a small set of architectures, each best suited to a different deployment regime. Second, the relative ranking of these pipelines is highly sensitive to geometric complexity: pipelines relying on the QGIS expression engine for per-frame calculations suffer rapidly degrading performance as dataset complexity increases, whereas pipelines that relocate the geometry-traversal cost to load time remain comparatively robust. *Refutation criterion:* the first prediction is refuted if a single condition is simultaneously the fastest and the lowest in memory on both datasets. The second is refuted if the performance-degradation rates (the slopes of the latency curves) are statistically indistinguishable across the six conditions. RQ1 also fails outright, as a matter of basic feasibility, if no condition reaches 30 FPS on either dataset — the frame rate below which an animation is no longer perceived as smooth [9]. We set this bar as an absolute frame rate rather than as a ratio against a baseline, because 30 FPS rests on published evidence about human motion perception, whereas the choice of any particular ratio would be arbitrary.

RQ2 — Live trajectory reconstruction from heterogeneous feeds. *Can a single Extract, Transform, Load (ETL) (extract-transform-load) pipeline ingest two fundamentally different real-time feeds (STIB-MIVB's stateless distance feed and De Lijn's GTFS-RT JavaScript Object Notation (JSON) feed), reconstruct* `tgeompoint` *trajectories from them, and sustain end-to-end ingest-to-database latency below the informal* 30*-second bar?*
The 30-second target is not a STIB-MIVB Service Level Agreement (SLA); it is an engineering envelope derived from the upstream feeds themselves: both operators publish at a 20-second polling cadence (Section 2.6), so any platform that adds more than ∼10 s of pipeline overhead becomes the dominant source of staleness rather than the feed itself.

The bar is the natural "don't make the data stale" threshold for this class of system. The metric reported is the *ingest-to-DB* latency: the wall-clock difference between the upstream feed's observation timestamp and the moment the corresponding `appendInstant` commits in MOBDB. Time-to-First-Pixel on the user-facing API is a design target for the platform but is not measured directly in this campaign (Section 7.15). *Hypothesis:* strict three-layer isolation (ingest, transform, load) plus a hot/cold table split for high-churn `tgeompoint` updates keeps ingest-to-DB latency bounded even when one feed degrades. *Refutation criterion:* median ingest-to-DB latency > 30 s on a multi-day window, or $p_{95} > 60$ s.

RQ3 — Web-based versus desktop visualisation at city scale. *At what trajectory count does a vector-tile-plus-WebGL web stack outperform a column-oriented QGIS plugin on the same hardware, dataset and time window, and what operational signals does the platform produce on the live Brussels feeds?*
Hypothesis on the cross-over: for $N \lesssim 1{,}000$ active trips the two stacks are screen-bound and indistinguishable; around $N \sim 1{,}000$ a clean cross-over appears; beyond $N \gtrsim 10{,}000$ only the web stack sustains 30 Frames Per Second (FPS). On the operational side we report the *volume and traceability* of typed alerts (proximity, slow speed, stuck) emitted into a single `rt.<feed>_alert` table over weeks of continuous operation; a precision/recall study against manually-annotated ground truth is acknowledged as out of scope for the present campaign and listed in Section 7.15. *Refutation criteria (joint):* either (i) the two stacks deliver indistinguishable medians across the nine scales tested under the caveat of their respective FPS semantics (Section 8.2), or (ii) the platform fails to emit typed alerts traceable against the literature thresholds over the multi-day deployment window. We do not phrase RQ3 as a one-sided "web > desktop" claim, because the FPS metric is not strictly commensurable across the two stacks.

These three questions are answered in Sections 6, 7 and 8 respectively, with the evaluation protocol of Section 5 applied uniformly across the three.

1.4 Contributions

This work makes four contributions:

- **A six-pipeline cross-architecture comparison** of animated MOBDB trajectory rendering in QGIS, on two contrasting datasets (∼5 000 trips each, sparse STIB at ∼14 vertices/trip and dense AIS at ∼1 230 vertices/trip). The comparison identifies a targeted optimisation of Manzer's published canvas-item architecture as the latency leader on both datasets (∼181 FPS STIB, ∼65 FPS AIS at $N = 5\,000$, i.e. 21.8× / 35.7× over the MOVE upstream baseline), a candidate opt-in memory-layer patch to MOVE as the deployable upstream contribution (16.1× / 4.4×, the asymmetry attributable to the QGIS expression engine), and a column-oriented MOBDB pre-sampling as an analytical option that trades ∼760 MB of RAM for 35 FPS on AIS. Seven invalidated claims and five methodological lessons from the iteration history are documented as a separate contribution. (Chapter 6.)

- **RTDataHub**, a three-layer operational platform that ingests STIB-MIVB and De Lijn 24/7, reconstructs trajectories from two by design different feeds, persists them as MOBDB `tgeompoint`, and serves them through a vector-tile API backed by a minimal MapLibre frontend. The platform has been running continuously on the live feeds for the duration of the writing of this thesis. Its trip reconstruction is additionally checked against a

single-session manual ground truth collected from operations-room cameras, reported in Section 7.5.7. (Section 7.)

- **A multi-class real-time alert detector** (proximity / slow speed / stuck) calibrated against operational thresholds from the transit-operations literature, with a freshness gate derived empirically from the measured staleness distribution of incoming feed positions. The detector emits typed alerts into a single `rt.<feed>_alert` table that is queryable and replayable. (Section 7.11.)
- **A head-to-head comparison** of the QGIS desktop pipeline and the RTDataHub browser frontend on nine scales of trajectory count, on the same machine and the same data, with measurement protocols designed so a future student can reproduce the experiment. (Section 8.)

1.5 Architectural Overview

Figure 1.1 gives a one-page architectural picture of the two tracks. The two contribution tracks converge on the same MOBDB store, which itself ingests the STIB-MIVB and De Lijn real-time feeds. Section 2 introduces the background needed to read this figure precisely.

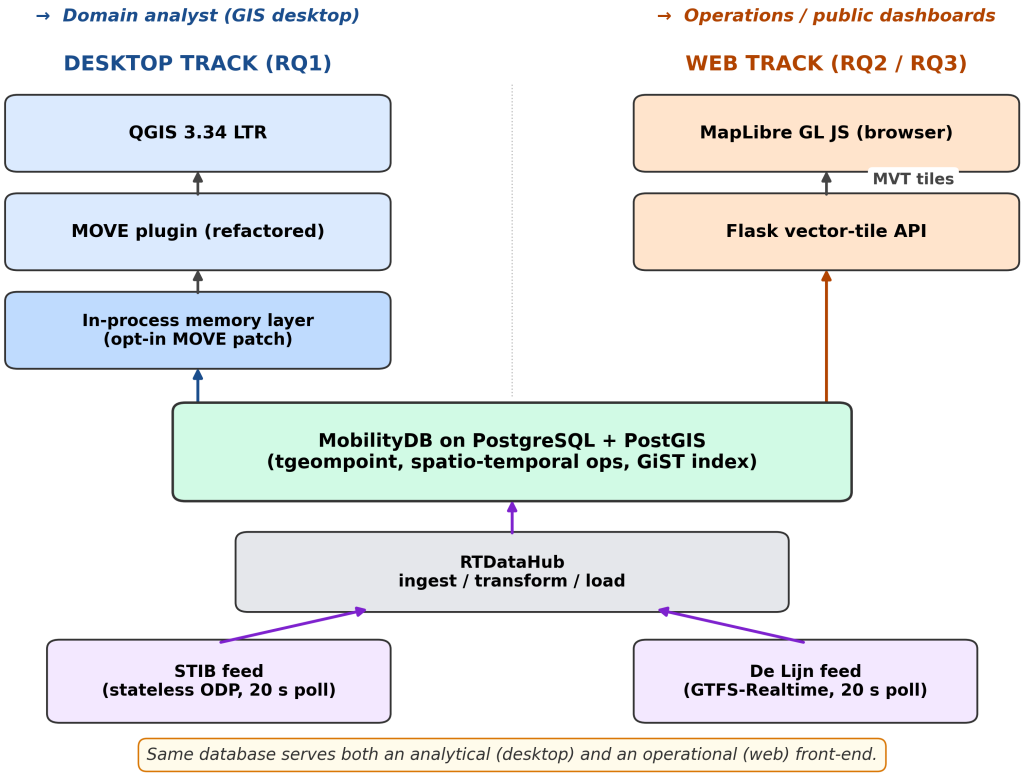


Figure 1.1: High-level architecture of the two contribution tracks of the present work. The desktop track refactors the QGIS MOVE plugin around an in-process memory layer, the opt-in upstream patch evaluated in Chapter 6 and benchmarked against the web stack in Chapter 8; two further tiers explored in Chapter 6, an in-process optimisation of Manzer's canvas-item architecture and a column-oriented NumPy pre-sampling, are not shown here. The web track is the RTDataHub platform. Both share a single MOBDB backend that ingests the STIB-MIVB and De Lijn feeds.

1.6 Thesis Structure

Section 2 introduces the formal moving-object model, the MOBDB type system, General Transit Feed Specification (GTFS)/GTFS-RT, indexing, the web-mapping stack and the operational definition of bus bunching. Section 3 quantifies the Brussels workload and makes the Manzer bottleneck explicit. Section 4 surveys related work and positions our contributions through three comparison tables. Section 1.3 states the research questions, and Section 5 the evaluation methodology used uniformly across RQs. Sections 6, 7 and 8 answer RQ1 (QGIS rendering pipelines), RQ2–RQ3 (RTDataHub) and the head-to-head comparison, respectively. Sections 9 and 10 discuss generalisability and conclude.

1.7 Disclaimer

The author wrote the manuscript and produced every measurement reported, with one exception stated where it occurs: the raw read-latency figures of the MEOS.js cross-check (Section 7.13.2) were produced by Dawid Banas and are reproduced there as external corroboration. For text revision, the author used the spelling and grammar checker Grammarly and an LLM-based writing assistant for editorial suggestions on phrasing and consistency; every sentence in the final manuscript was read and accepted by the author. No content was generated without author oversight. Code, data and figure-generation scripts are the author's own, except where explicitly attributed to the cited upstream projects (MobilityDB [1], MEOS, MapLibre, etc.). Where this work re-uses sentences from earlier reports written by the author for the same supervisor (notably the intermediate "RQ2 detailed report" delivered during the year), they are integrated and updated to match the present numbers and structure.

The platform RTDataHub described in Section 7 runs on a single virtual machine and operates on public real-time feeds; it processes no Personally Identifiable Information (PII). The GDPR positioning of the system is discussed in Section 9.

Chapter 2

Background and Preliminaries

This chapter gives a CS reader, with no prior exposure to mobility data, the vocabulary needed for the rest of the thesis. We cover: the *moving-object* model (§2.1), MOD formalism (§2.2), the MOBDB type system (§2.3), the two transit-feed formats we ingest (GTFS §2.4, GTFS-RT §2.5, plus the non-standard STIB-MIVB variant §2.6), spatial indexing (§2.7), the web mapping stack (§2.8), and what bus bunching means in practice (§2.9). Readers already familiar with MOBDB and GTFS can skip directly to Section 3.

2.1 Trajectories and Moving Objects

A *moving object* is anything whose location changes over time. In this work the moving objects are buses, trams and metros. Formally, the position of one moving object is a function

$$p\colon T \to S \tag{2.1}$$

that maps an instant $t \in T$ to a point $p(t) \in S$, with $S = \mathbb{R}^2$ (or a 1-D curve parameterised by distance). A *trajectory* is the graph of such a function over a finite time window [3].

In practice, sensors emit observations at discrete instants. The *abstract* model treats p as continuous. The *discrete* representation stores the samples $\{(t_i, p(t_i))\}_{i=1}^{n}$ and interpolates between consecutive ones. MOBDB defaults to linear interpolation:

$$p(t) = p_i + \frac{t - t_i}{t_{i+1} - t_i}\,(p_{i+1} - p_i) \quad \text{for } t \in [t_i, t_{i+1}]. \tag{2.2}$$

Linear interpolation is fine at our feed rates (20 s for both STIB-MIVB and De Lijn). In Section 7.7 we interpolate *along the polyline of the scheduled GTFS shape* instead of straight in $\mathbb{R}^2$, so that interpolated positions do not cut through the Brussels canal.

The benefit of treating p as a first-class type is that queries like "where is bus b at time t?" or "which pair came within 50 m between 07:00 and 09:00?" are expressed and planned by the database, not reimplemented in the application.

2.2 Moving Object Databases

A MOD (Moving Object Database) exposes the continuous function p as a first-class data type [3]. The framework rests on three pillars: (i) *type constructors over time*, each base type τ has a temporal counterpart *moving*(τ) whose values are functions from time to τ; (ii) *lifted operators*, spatial operators (distance, intersect, ...) are extended so they take and return moving values; (iii) an *Abstract Data Type (ADT) interface* that exposes the new types to the host

Database Management System (DBMS) (PostgreSQL here) with native syntax, index support and query planner integration.

The first open-source production-grade implementation is MOBDB [1], an extension to PostgreSQL/PostGIS. Its C-level engine Mobility Engine Open Source (MEOS)[1] can be linked from non-database hosts; the Python binding is Python binding for the MEOS library (PyMEOS).

2.3 MobilityDB Type System and Key Operators

MOBDB provides a discrete representation of the abstract moving-object model of Section 2.1. The type that matters for this thesis is the moving geometric point, written `tgeompoint` (a moving point in space and time). It can take three structural forms:

`tinstant` One observation: a point plus a timestamp. Notation: `Point(lon lat)@T` where `T` is an International Organization for Standardization (ISO)-8601 timestamp. Example: `Point(4.4057 50.8163)@2026-04-28 18:24:19+00`.

`tsequence` A continuous sequence of `tinstant` values, with explicit lower and upper bounds on the time interval, an interpolation rule (linear by default), and inclusivity flags on the bounds. Notation: `[Point(`x_1 y_1`)@`T_1`, Point(`x_2 y_2`)@`T_2`,...]`. This is the central form: *one trip* of one vehicle is stored as one `tsequence`.

`tsequenceset` A set of `tsequence` values with disjoint time intervals. Used when the observed trajectory has gaps. E.g., a vehicle that goes off-feed during a layover and reappears later.

The default interpolation rule is the linear rule of Equation 2.2.

Interpolation flag used throughout this thesis. MOBDB supports two interpolation flags on a `tgeompoint`: `LINEAR` (the default) and `STEP`. Throughout this thesis we use the `LINEAR` flag, consistent with sub-second vehicle dynamics: between two timestamped instants the moving point is assumed to follow a straight line in space, which matches what `tsample`, `valueAtTimestamp`, and `atPeriod` implicitly assume on the trajectories we ingest. The `STEP` flag — which would freeze the position between two updates — is appropriate for piecewise-constant signals (e.g. "bus is at stop s") but not for continuous motion, and is not used here.

Operators used in the present work. Five operators from MOBDB's algebra are used everywhere:

`atPeriod(`T**`,`** $[t_a, t_b]$**`)`** restricts a temporal value to a time interval; called on every time-scrub.

`valueAtTimestamp(`T**`,`** t**`)`** returns $p(t)$, with interpolation; this is the per-frame call of the row-oriented baseline [2], replaced by bulk `tsample` in Section 6.

`minDistance(`T_1**`,`** T_2**`)`** returns the minimum spatial distance between two moving values, computed analytically from the linear pieces; this is the foundation of the bunching detector (Section 7.11).

`appendInstant(`S**`,`** i**`)`** returns a new `tsequence` that extends S with instant i; the main write operator on the hot table.

`tsample(`S**`,`** Δt**`)`** samples S at regular timestamps; called once at start-up to build the column-oriented matrix of Section 6.

[1] https://github.com/MobilityDB/MEOS

2.4 General Transit Feed Specification (GTFS)

GTFS is the de facto international standard for publishing static public-transport schedules [10, 11]. A GTFS *feed* is a ZIP archive containing a small set of Comma-Separated Values (CSV) files with a fixed schema. The tables we use are listed in Table 2.1.

Table 2.1: Static GTFS files used by this thesis. Many other files exist in the specification (fares, transfers, frequencies, ...) but are not consumed by RTDataHub.

File	Role	Used in this thesis
`routes.txt`	The set of bus / tram / metro lines, with type, short name and long name.	line catalogue, frontend selector
`trips.txt`	Each trip = one execution of a route in one direction at one time, with a `shape_id`.	link from a live tick to a shape
`stops.txt`	Stops with location and metadata, including which are line terminals.	"stuck" alert exclusion zones
`stop_times.txt`	Per-trip planned arrival / departure times at each stop.	planned segment speeds for the slow-speed detector
`shapes.txt`	Polyline geometry of each scheduled trip, sampled as $(lat, lon, shape\_dist\_traveled)$ points.	STIB position interpolation
`calendar.txt`	Day-type schedule (weekday, Saturday, Sunday, public holiday).	day-of-week matching

The `shape_dist_traveled` field deserves a quick comment, since it is central to the STIB-MIVB ingestion pipeline. For each vertex of a trip's polyline, it gives the distance *along the polyline* from the first vertex. STIB-MIVB reports vehicle positions as "distance d past stop s" rather than as a (lat, lon) pair (see Section 2.6). To translate this into a (lon, lat), we walk the polyline of the trip's shape until the cumulative `shape_dist_traveled` matches the reported distance, then linearly interpolate the geographic point.

2.5 GTFS-Realtime

GTFS-RT [4] extends static GTFS with three real-time feed types: `TripUpdate` (estimated arrival times), `VehiclePosition` (instantaneous lat/lon, bearing, speed) and `Alert` (human-facing service disruptions). The De Lijn feed used here is a `VehiclePosition` feed, served as GTFS-RT-shaped JSON (the wire format negotiated with the De Lijn API) over HyperText Transfer Protocol (HTTP). We poll it at a configurable interval (default 20 s, the same rate as the STIB-MIVB ingestor). A `vehicle_id` field is stable across consecutive ticks, so trip identity is trivially recoverable.

2.6 The STIB-MIVB Open Data Feed

STIB-MIVB does *not* publish a GTFS-RT-conformant `VehiclePosition` feed. Instead, the operator's Open Data Portal (STIB-MIVB) (ODP) exposes a custom JSON feed with the following per-vehicle structure (anonymised example; field names match the actual feed):

In other words, for each vehicle in service the feed reports: the line being operated, the direction (encoded as the destination terminal stop), the last stop the vehicle passed (`pointid`), and the curvilinear distance in metres along the route since that stop. There is **no** GPS (lat, lon) pair, and **no** stable vehicle identifier across ticks. Both have to be reconstructed by the ingestion

```
{
  "line_id":           71,
  "direction":         2596,
  "point_id":          8762,
  "distance_from_point": 235
}
```

Listing 2.1: Sample JSON payload for transit positioning

pipeline (Section 7.7) from the GTFS shape lookup of Section 2.4 and a greedy spatial matching procedure documented in Section 7.8.

The polling rate of the STIB-MIVB feed is set by the operator's API terms; in our deployment we poll every 20 s.

2.7 Spatial and Spatiotemporal Indexing

PostGIS [12] provides two index types: `GiST` (R-tree-style: a balanced tree of bounding rectangles, supporting range and nearest-neighbour queries by tree descent [13]) and `SP-GiST` (space-partitioning, effective on clustered points). MOBDB extends `GiST` to spatiotemporal bounding boxes, so it can prune in space and time at the same time [1]. RTDataHub uses one composite `GiST` index per hot trip table. Index choice is otherwise outside the scope of the present work: the writer-side hot/cold split of Section 7.8 dominates.

2.8 Web Mapping Stack: MapLibre, MVT, Flask

The web frontend used in Section 7.12 is a thin combination of three open-source components.

Mapbox Vector Tile (binary tile format) (MVT) (Mapbox Vector Tile) [14] stores vector geometry per (x, y, z) tile in a Protobuf payload, leaving styling to the client; it is the de facto standard for streaming interactive web maps over HTTP. MapLibre [6] is the open-source successor of Mapbox GL JS, a Web Graphics Library (browser-side 3D/2D GPU API) (WebGL) renderer that consumes MVT. It supports temporal styling via *expressions* on feature attributes, which turns time scrubbing into a paint-expression update rather than a re-fetch. Flask [15] serves the tiles via one endpoint (`GET /tile/{z}/{x}/{y}.mvt?at=`T) that compiles into a single MOBDB Structured Query Language (SQL) statement using `atPeriod` and `valueAtTimestamp`.

2.9 Bus Bunching

Bus bunching is a well-studied transit-operations phenomenon where two or more vehicles of the same line, meant to be evenly spaced, end up running close together [16–20]. This produces a service gap that propagates: passengers behind the cluster wait longer than scheduled, the trailing bus loads up, slows further, and the cluster grows.

Operational thresholds. The proximity detector of Section 7.11 flags pairs whose measured gap falls under a threshold sized as a *fraction of the nominal spacing of the line at the current hour*. The per-line nominal spacing is $f \cdot v$, using the scheduled headway f and a mean cruise speed v derived from GTFS. The fractions, the threshold clamp, and the cascade structure are stated and justified in Section 7.10. Two further detectors (slow-speed and stuck) live in the same architectural slot and emit alerts of the same shape; they are described in Section 7.11.5.

2.10 Tools and Environment

QGIS and the MOVE plugin. QGIS [5] is the open-source desktop GIS we use for analyst-side visualisation. The MOVE plugin, designed by Manzer [2] and refactored in Section 6, is a Python extension to QGIS. It fetches MOBDB trajectories, animates them through QGIS' Temporal Controller, and exposes an attribute panel synchronised to the time slider.

Python data handling. The two scientific Python libraries we lean on are NumPy [21] (dense array operations, central to the column-oriented refactor of Section 6) and pandas [22] (timestamped dataframes for the offline benchmarks of Section 5). For the live ingestion pipeline of Section 7.4, we use the `psycopg` driver to talk to PostgreSQL and the `requests` library to talk to the HTTP APIs.

Hardware. Two distinct virtualised environments are used in this work. The *production deployment* of RTDataHub runs on an Ubuntu Virtual Machine (VM) provided by Bruxelles Mobilité: 4 vCPUs (Intel Xeon Gold 6426Y, Sapphire Rapids), 7.8 GiB of RAM, 8 GiB of swap, a 300 GB SCSI virtual disk backed by host Solid-State Drive (SSD) storage, and no Graphics Processing Unit (GPU) exposed to the guest. This is where the platform has been running on the live feeds for several weeks; the end-to-end latency figures of Section 7.13.1 reflect this environment.

The *benchmark machine*, used for the RQ1 column-oriented refactor (Section 6) and the head-to-head comparison (Section 8), is a separate Ubuntu 25.04 VMware VM. The host has an Intel Core i5-11400F (Rocket Lake, 6 cores / 12 threads, 2.60 GHz). The guest sees 4 vCPUs (hyperthreading not exposed), 7.2 GiB of RAM, 3.8 GiB of swap, and a 500 GB VMware virtual disk (host SSD-backed but advertised to the guest with `rotational=1`; `mq-deadline` I/O scheduler). The hypervisor exposes only a VMware SVGA II framebuffer, with no discrete GPU and no GPGPU/CUDA path, so MapLibre WebGL runs through Mesa software rendering. Section 5.1 pins the exact software versions (PostgreSQL 17.9 + PostGIS + MOBDB compiled locally with gcc 14.3, QGIS 3.34 LTR, Python 3.13).

Section Summary

This chapter introduced the conceptual machinery used in the rest of the thesis. The two key abstractions are the *moving object* of Equation 2.1, a function from time to space — and its MOBDB discrete representation as a `tgeompoint`, structured as `tinstant`, `tsequence` or `tsequenceset` (Section 2.3). Five operators are central: `atPeriod`, `valueAtTimestamp`, `minDistance`, `appendInstant` and `tsample`. The two real-world feeds we ingest, STIB-MIVB and De Lijn, have very different shapes: STIB-MIVB gives a stateless distance along a shape, with no GPS and no vehicle id; De Lijn is GTFS-RT with direct GPS and a stable id. Section 7.7 will normalise them into the same `tgeompoint` representation. Finally, the operational definition of bus bunching is a cascade of three strict thresholds on the ratio of measured to nominal spacing, clamped to $[20, 500]$ m. With these prerequisites in place, the next section quantifies the Brussels workload and makes Manzer's bottleneck explicit.

Chapter 3

Problem Statement

The central problem is the **lack of scalability** of the current integration between MobilityDB and QGIS, more generally, between a rich spatiotemporal database and the user-facing visualisation layer. This section makes that claim precise. We start from Manzer's two architectures (Sections 3.1.1 and 3.1.2). We then examine why the obvious next move (web vector tiles) does not resolve the problem (Section 3.2), describe the operational context (Section 3.3) and state the resulting gap (Section 3.5).

Figure 3.1 contrasts Manzer's two architectures — Interactive Mode and TimeDeltas Mode — with the structural requirement this thesis sets out to meet, and makes visible which resource each existing mode saturates.

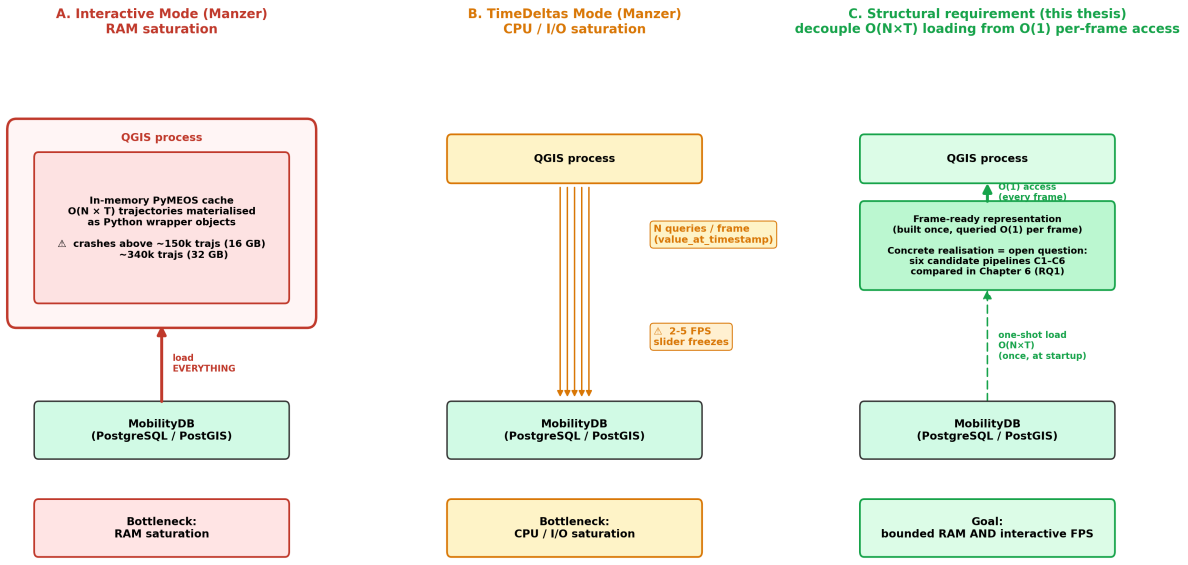


Figure 3.1: Manzer's two architectures and the structural requirement this thesis pursues. (A) Interactive Mode saturates RAM; (B) TimeDeltas Mode saturates the CPU and the I/O path; (C) the requirement is a frame-ready representation that decouples the one-shot $O(N \times T)$ load from $O(1)$ per-object access, holding bounded RAM and an interactive frame rate together — the six candidate pipelines that realise it are compared in Chapter 6. Dashed arrows mark one-shot $O(N \times T)$ loads, solid arrows mark cheap $O(1)$ per-object accesses.

3.1 Structural Failures of Previous Architectures

Manzer [2] explored two architectures for shipping MobilityDB data to QGIS: *Interactive Mode* and *TimeDeltas Mode*. Both were functionally correct: animations ran, the map updated, the temporal controller behaved, yet both collapsed against physical limits inherent to their design.

3.1.1 RAM Saturation in Interactive Mode (In-Memory Monolithic Load)

Interactive Mode uses PyMEOS [23] (the Python bindings to MobilityDB's C kernel) to eagerly load the full dataset into QGIS. Every `tgeompoint` (a moving point in space and time) column is pulled as Python wrapper objects, every instant is materialised, and the Temporal Controller iterates over this in-memory structure. The appeal is simplicity: once everything is in RAM, each frame is a cheap lookup.

The bottleneck is **space complexity**. Memory grows linearly in the total number of trajectory points. Even with 32 GB of RAM, loading the full STIB corpus (≈6 GB raw, inflated to 12–18 GB of Python-wrapped objects in QGIS) causes immediate saturation and crashes [2, p. 48]. The cause is Python-side overhead: PyMEOS wrappers, Python lists, QGIS features and the rendering cache each add a layer of indirection. Manzer reports a 2–3× inflation over raw MobilityDB storage [2, p. 48], in line with the typical cost of moving from packed columnar representations to per-object Python structures [21, 22].

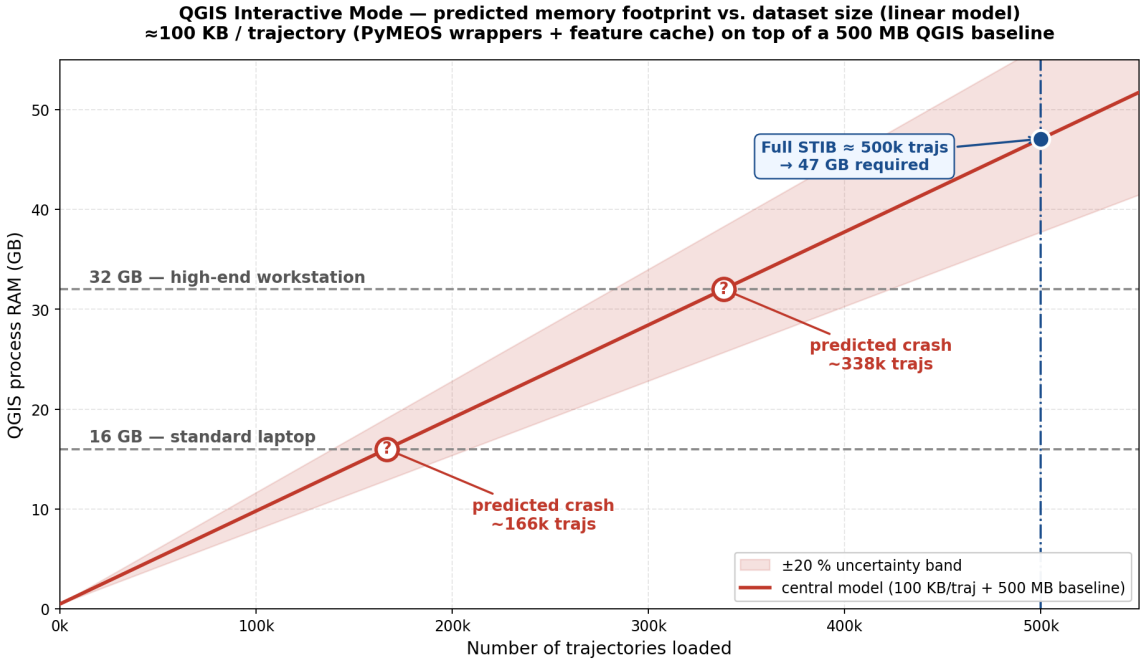


Figure 3.2: Memory footprint of QGIS Interactive Mode as a function of the number of loaded trajectories, under a *linear extrapolation model* (not a direct measurement) of ≈100 KB per trajectory on top of a 500 MB QGIS baseline, calibrated on the per-trajectory inflation reported by Manzer [2, p. 48]. The two crash points where the model crosses the hardware ceilings (≈166k and ≈338k trajectories) are extrapolated from the same model, not measured points; the horizontal dashed lines mark the two hardware profiles from Section 5 and the vertical dash-dotted line marks the size of the full STIB corpus, which sits beyond both profiles.

3.1.2 Latency and CPU Bottlenecks in TimeDeltas Mode (Dynamic Querying)

To sidestep the memory issue, Manzer's TimeDeltas Mode loads data on demand in short temporal chunks: each frame queries only observations in the current window, renders them, and discards them. Peak memory stays bounded by a single chunk (a few MB for a short window on STIB).

The design moves the bottleneck rather than removing it. **I/O** carries one PostgreSQL round-trip per frame plus per-round-trip overheads (network latency, TLS, query parsing, planning, spatial-index descent). On a local laptop MobilityDB, query latencies on the STIB working set hit a few hundred milliseconds [2, p. 57]: enough to drop the frame rate to 2–5 FPS and freeze the slider. **CPU** is loaded by `value_at_timestamp` (a MobilityDB function that interpolates a moving object's position at a given instant), which MobilityDB evaluates on every queried `tgeompoint` to interpolate the position at the requested instant. Each evaluation is cheap, but one per object per frame multiplies to hundreds of thousands per second at city scale. QGIS's feature cache does not understand continuous temporal semantics, so it cannot reuse results across frames.

The net effect [2, pp. 54–60] is that TimeDeltas trades a hard failure (crash) for a soft failure (frozen UI). This is arguably worse for user perception, since the user is left unsure whether the app has crashed or is just slow, producing the exploratory-strategy degradation Liu and Heer [24] documented empirically.

3.2 Why Vector Tiles, by Themselves, Are Not the Answer

Vector tiles [6, 14] (pre-cut map fragments served on demand, like Google Maps) are not, by themselves, the answer inside QGIS. We separate two distinct objections, because conflating them would overstate the limitation of vector tiles in general.

Pragmatic objection (QGIS-client limit). The QGIS vector-tile API does not support dynamic updates to a tile source URL without rebuilding the layer [2, Chapter 5], so a time-parameterised URL defeats tile caching. This is a software limit of the QGIS client, not of the vector-tile format. A different client (a browser with MapLibre, Chapter 7) does not suffer from it, and indeed our web track *does* use vector tiles successfully.

Structural objection (tile-format limit for trajectories). Tiling assumes that *geometry* is stable; for trajectory data, geometry *is* the time-varying quantity, and merging per-tile temporal attributes with static geometry forces vertex-buffer rebuilds that wipe out the bandwidth advantage past a few hundred thousand objects [25]. This is a real limitation of MVT as currently specified — and partly addressed by the column-oriented MapLibre Tile (MLT) successor [26] discussed in Section 4.4. For live trajectories, a workable design needs server-side per-frame tiling *plus* client-side GPU interpolation, which a naive PgTileServ + QGIS does not provide; the RTDataHub web stack of Chapter 7 reproduces this pattern with a custom Flask tile endpoint instead.

3.3 Operational Constraints from the Brussels Feeds

Two properties of the Brussels feeds sharpen the architectural discussion.

The STIB open-data feed [7] is *stateless*: at a 20 s polling cadence it reports the current odometric position (distance travelled along a planned route) of every vehicle on its planned shape, but no trajectory history and no vehicle-level identifier stable across sign-offs. Any architecture wanting continuous trajectories (not independent dots) must reconstruct them *server-side* by

appending each observation to a running time-indexed geometry per vehicle run. This applies classical linear-referencing primitives (projection onto a planned shape, cumulative-distance ordering) without the heavier HMM map-matching machinery surveyed by [27]. Client-side reconstruction would require a per-run cache that does not scale across tabs, sessions or users.

The De Lijn GTFS-Realtime feed [4, 8] (Google's standard format for live transit data), by contrast, is state-aware but heterogeneous: `VehiclePosition` messages with geodetic coordinates and per-trip identifiers, polled at the same 20 s cadence as STIB but with different coverage and field semantics. A platform has to accept both shapes, both identifiers and both failure modes, and expose a unified trajectory abstraction downstream, which is precisely what motivates RTDataHub.

3.4 Quantitative Profile of the Brussels Workload

Table 3.1 summarises the workload that the platform must absorb. The volumes are well within a single PostgreSQL / PostGIS / MOBDB instance [1, 12], but already crash Manzer's in-memory QGIS plugin on a single running-day subset [2, p. 48]. Scaling the in-memory architecture to a full year is therefore a losing strategy; both tracks of the present work start from that observation.

Table 3.1: Quantitative profile of the Brussels workload that RTDataHub absorbs. The 1,835,566 STIB count is measured over the 24 h window of Table 7.10; the De Lijn count is over the same window after the Brupass XL boundary filter (Appendix D); the concurrent-trip count is the peak observed in the live frontend over the 2026-04 capture window.

Metric	**STIB-MIVB**	**De Lijn**
Position observations / day	$\sim$ 1,836 k	$\sim$ 608 k
Indexed footprint / year (estimate)	$\sim$ 200 GB	$\sim$ 75 GB
Peak concurrent active trips (weekday peak)	17,118	$\sim$ 3,000
Polling cadence, positions (default)	20 s	20 s
Polling cadence, editorial alerts	600 s	20 s (piggyback)

3.5 The Gap This Work Closes

Combining the above: given MobilityDB as the server and a choice between a desktop (QGIS) or web client, current paradigms force a choice between three bad options. **(a)** visualising small amounts of data smoothly (Interactive Mode), **(b)** visualising large amounts with prohibitive latency (TimeDeltas Mode), or **(c)** reaching for an off-the-shelf web map (deck.gl, kepler.gl) that bypasses the database entirely and forces analytical workflows back into a row-flat layer model, losing the temporal-algebra and ad-hoc-PostGIS power that motivates picking MobilityDB in the first place. No architecture in the MobilityDB–QGIS ecosystem streams large mobility data without maxing out client RAM or CPU, and, to our knowledge, no prior open web visualisation couples MobilityDB to the live STIB and De Lijn feeds. This work closes both halves in parallel: re-engineering the desktop pipeline so the per-frame cost no longer scales with trajectory length, and building a web-based RTDataHub platform that runs continuously on Brussels feeds with MobilityDB as trajectory store.

Chapter 4

State of the Art

The scalability problem from Section 3 sits at the intersection of moving-object databases, spatio-temporal indexing, web mapping, progressive visual analytics, and real-time transit monitoring. This chapter surveys each body of work and connects it to the thesis's contributions.

4.1 Moving-Object Databases and Continuous Temporal Types

The conceptual foundation of our work is the *moving-object database* (MOD) paradigm, introduced by Güting and Schneider in their book *Moving Objects Databases* [3]. It was later extended through a long line of papers on spatio-temporal pattern queries [28] and applications [29]. The core idea is to promote continuously-varying geometries to first-class data types. A `moving point`, for example, is not a sequence of rows but a single value that represents a trajectory as a piecewise-linear function from time to space. Classical database operators (`SELECT`, `JOIN`, `AGGREGATE`) are then extended to work on these temporal types. New operators are added too, such as `value_at_timestamp`, `duration`, `length`, and the `&+` append operator used everywhere in the present work.

MobilityDB [1] is the open-source embodiment of that paradigm on top of PostgreSQL and PostGIS [12]. It introduces a family of temporal types (`tgeompoint`, `tfloat`, `tbool`, `ttext`) and an algebra of several hundred operators covering interpolation, projection, ever/always predicates, relative-position tests, and spatiotemporal joins. The reference textbook by Zimányi et al. [30] gives a complete catalogue of those operators. For interaction from Python, PyMEOS [23] provides a thin wrapper around the MEOS C library and exposes the same type hierarchy to Python code, including the QGIS plugin environment. MobilityDB inherits from PostGIS all the benefits of the R-tree and its variants [13, 31]. It then adds spatiotemporal bounding boxes that extend R-tree reasoning to the time dimension.

MobilityDB is not the only MOD. Academic systems like Hermes and SECONDO explore similar ideas in other ecosystems. The thesis does not target them directly, but they share the same conceptual basis and the same last-mile visualisation gap. Our design choices — reconstruction server-side, only pre-aggregated views on the client — would transfer to any MOD that implements the continuous-type paradigm.

Table 4.1 positions the surveyed MOD systems on four capability axes that matter for this work: native temporal types, the operator algebra (lift, projection, predicates), distributed execution, and a streaming-ingest path. MOBDB is the only system that combines the three first-class features — native temporal types, a full operator algebra and a streaming-ingest path — at production maturity. None of the MOD systems were designed with sustained streaming in mind; that mismatch motivates the engineering work in Section 7.

Table 4.1: Comparison of moving-object database systems on the four axes relevant to this thesis (✓ = native, ✗ = absent, ○ = partial / experimental).

System	Temporal types	Operator algebra	Distributed	Streaming ingest
SECONDO [3]	✓	✓	○	✗
Hermes (Oracle layer) [32]	✓	○	✗	✗
HadoopTrajectory [33]	○	○	✓	✗
TrajSpark [34]	○	○	✓	○
MobilityDB [1]	✓	✓	○ (Citus)	○
MEOS / PyMEOS [23]	✓	✓	✗	○

4.2 Spatial and Spatio-Temporal Indexing

Classical spatial indexing, on which PostGIS and MobilityDB are built, starts with Guttman's R-tree [13] and its variants (R^+-tree [31], R*-tree, Hilbert R-tree). Spatio-temporal extensions go back to Pfoser et al. [35]: the general principle is to extend the MBR (minimum bounding rectangle) into a minimum bounding volume (MBV) that also covers time, so temporal range queries can be accelerated by tree descent.

The two spatio-temporal indexing papers closest to this work are Hojati et al.'s **DSTree** [36] and Zhao et al.'s **LevelDBST / XS2** [37]. They offer contrasting answers to the same question, "how do I index billions of trajectory observations efficiently?", and each captures one of the two workloads we care about.

4.2.1 Structure-Based: DSTree

Hojati et al.'s **DSTree** (Distributed Spatio-Temporal Tree) targets distributed networks of geolocated sensors [36]. It uses a dual-level architecture: a top-level Interval Tree over time ranges supports Allen-style temporal operators (overlaps, during, meets) in $\mathcal{O}(\log n)$. Each leaf points to a spatial QuadTree for the associated extent. Separating the dimensions keeps each subtree simple and balanced, at the cost of a complex maintenance protocol under high-frequency updates — where write-heavy streaming workloads strain it.

4.2.2 Encoding-Based: LevelDBST and XS2

Zhao et al.'s **LevelDBST** [37] takes the opposite approach. Instead of a tree, it encodes spatial and temporal coordinates into keys of a key-value store, designed so that neighbouring trajectories cluster together. Live trajectories use a *Dual-Key Encoding (DKE)* that keeps writes cheap. Static trajectories use **XS2**, an extension of the S2 Hilbert-curve encoding. *AdaptSeg* breaks long trajectories into segments that fit neatly into index cells, minimising the false positives typical of Hilbert-curve indexing.

The DSTree-vs-LevelDBST contrast maps onto our two tracks. The desktop track (Section 6) is read-heavy on a moderate snapshot, so tree-based fits. The web track (Section 7) is a write-heavy streaming ingest, so encoding-based would fit better. For our deployment, MobilityDB's stock GiST index has proven sufficient.

4.3 Map Matching and Trajectory Reconstruction

A distinctive feature of the Brussels STIB feed is that vehicle positions are reported as *odometric distances along a planned shape* rather than raw GPS fixes. This sidesteps the hardest

problem in trajectory analysis: *map matching*, i.e. associating each raw GPS observation with a specific road segment. The wider map-matching literature is still central for our treatment of the De Lijn feed (which exposes raw geodetic coordinates) and for justifying the linear-referencing approach we apply to STIB.

Quddus et al. [27] survey the field, distinguishing geometric, topological, probabilistic and HMM/Kalman-filter methods. The Hidden Markov Model approach of Newson and Krumm [38] is the de-facto reference for noisy, sparsely-sampled traces. It models the road network as a hidden state and GPS as noisy emissions, and reliably recovers the most likely road sequence even at >1-minute sampling. Big-data variants [39] scale it to fleet-level workloads.

RTDataHub (Section 7) does not run a full HMM matcher: STIB already carries odometric positions, and De Lijn snaps onto its GTFS shape. We borrow the standard linear-referencing recipe of transit operations research, project every observation onto a known polyline (here the GTFS `shapes.txt` of the current trip) and use cumulative distance as the primary ordering key. Scheepens et al. [40] apply the same projection step but for *density maps* rather than trajectory reconstruction; we share the projection primitive, not the end-goal.

4.4 Next-Generation Vector Tiles: Row- Versus Column-Oriented

Vector tiles, popularised by the Mapbox Vector Tile (MVT) specification [14], are the industry standard for geospatial data on the web. MobilityDB emits MVT through PostGIS's `ST_AsMVT`, and every major web-map library (MapLibre [6], Leaflet [41], deck.gl [42], kepler.gl [43]) consumes it. MVT shows its age on Big Data workloads. Tremmel and Zink's **MapLibre Tile (MLT)** [26] frames the evolution as a row-to-column paradigm shift. MVT stores features row by row, so reading a single attribute (e.g. a timestamp) across all features forces a full parse. MLT groups data by column in line with Apache Parquet, allowing dictionary, run-length and delta encodings that shine on repetitive columns (trip IDs, timestamps).

Their benchmarks report up to 6× better compression and 3× faster decoding than MVT [26]. For our workload — tiles with few geometries but many time-varying attributes per geometry — the column-oriented layout is a good fit. An MLT-based connector would substantially reduce the network- and decoding-side latency that plagued Manzer's TimeDeltas mode [2]. We have not built one here (both RTDataHub and the refactored plugin use MVT), but the MLT paper is the main argument that the tile-size side of the performance equation should no longer be treated as fixed.

Table 4.2 summarises the headline differences between the two formats as reported by Tremmel and Zink [26].

4.5 Architectures for Large-Scale Web Visualisation

Large-scale web visualisation has converged on a server-thin-client pattern: pre-compute server-side, render GPU-side on the client. deck.gl [42], kepler.gl [43] and MapLibre [6] all follow it, using WebGL shaders to move per-frame cost off the Central Processing Unit (CPU). Johnson et al. [25] argue the same in the energy domain. Andrienko and Andrienko [44] supply the visual-analytics rationale: massive movement datasets reward aggregation more than raw-point rendering. The RTDataHub frontend follows this pattern directly, a minimal MapLibre client over a Flask vector-tile API, with every non-trivial computation (interpolation, thresholds, bunching) server-side. A more recent shift questions even that split: WebAssembly now lets compiled C

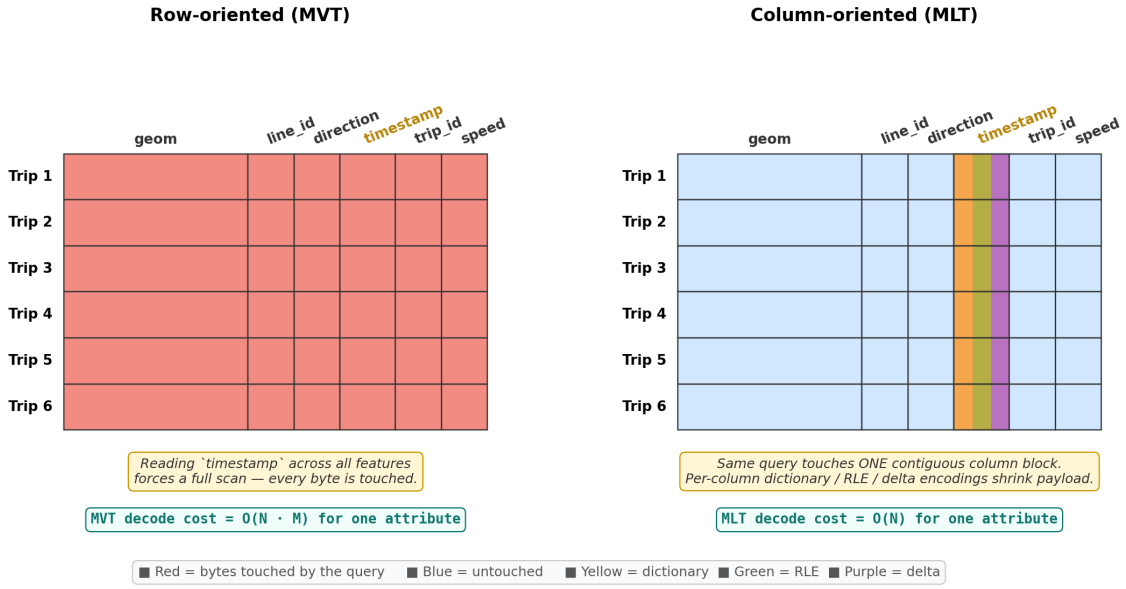


Figure 4.1: Row-oriented (MVT) versus column-oriented (MLT) data layout for vector tiles. Red cells mark the bytes a single-attribute query must scan: the MVT matrix is touched in full, whereas MLT touches only the target column, which can additionally be compressed with dictionary, run-length or delta encodings. The same column-oriented principle is applied in-memory in Section 6.

Dimension	MVT (Mapbox Vector Tile)	MLT (MapLibre Tile)
Storage layout	Row-oriented (feature by feature)	Column-oriented (attribute by attribute)
Compression strategy	Protobuf varint + zigzag	Dictionary, delta, RLE, FOR per column
Typical compression ratio vs. raw	$\sim 3\times$	Up to $\sim 6\times$
Typical decoding speed (relative to MVT)	$1\times$ (baseline)	Up to $\sim 3\times$ faster
Ecosystem maturity	Very mature, stable since 2015	Emerging, SIGSPATIAL'25
Best fit for	Static geometries	Attribute-heavy, dynamic workloads
Used in this work	Yes (RTDataHub v1)	Recommended as future work

Table 4.2: Headline comparison of the MVT and MLT vector-tile formats, adapted from Tremmel and Zink [26]. The column-oriented layout of MLT is the same design pattern we apply in-memory in Section 6.

and C++ run in the browser at near-native speed, so spatio-temporal operators that currently execute server-side could in principle move onto the client. MEOS.js, a WebAssembly port of the MEOS library behind MOBDB, is one such effort; Section 7.13.2 reports a read-latency cross-check against it, and Section 11 returns to the prospect of running the temporal-geometry operators inside the browser.

4.6 Progressive Visualisation and Interactive Analytics

Progressive Visualisation replaces "wait until fully loaded" with rapid partial results and user steering [45, 46]. The 500 ms latency budget we use in Section 5 is not arbitrary: Card et

al. [47] and Liu and Heer [24] show that exploratory strategies degrade past that threshold. A limited form of steering appears in Section 6, where we pre-sample only the active Temporal Controller window.

4.7 Vector Fields, Density Maps, and Visual Abstraction

Scaling visualisation is partly about *changing the abstraction*: 500 000 raw points cause visual clutter that defeats the purpose [44]. Density maps [40], cartograms [48] and movement-specific aggregations [49] all "aggregate first, then render". Li et al. [50] push this further with a travel-vector grid feeding a WebGL particle system whose render cost is independent of vehicle count. RTDataHub's bunching view applies the same lesson by reducing raw positions to a small set of alert markers.

4.8 Real-Time Public Transport and Bus-Bunching Detection

RTDataHub monitors headway regularity (the time gap between consecutive vehicles) and surfaces *bus bunching* events, two consecutive vehicles on the same line arriving within a small window. Newell and Potts [16] gave the first formal analysis: bunching is a natural instability of fixed-frequency lines. A late bus collects more passengers and slows further, while the bus behind catches up with a lighter load. Later work refined both analysis and countermeasures: Daganzo [17] and Eberlein et al. [18] on holding control, Bartholdi and Eisenstein [19] on self-coordinating routes, Moreira-Matias et al. [20] on online learning.

A key parameter is the *headway threshold*. A fixed threshold (e.g. 3 min) is simple but misleading. On a 4-min rush-hour line, 3 min is near nominal, while on a 20-min off-peak line it is clear bunching. Feng and Figliozzi [51] and Cats [52] argue that frequency-aware thresholds are necessary for operational relevance. In Section 7 we implement an adaptive threshold set to a fixed fraction of the planned headway at the current time of day, with a clamped fallback when schedule information is missing.

4.9 QGIS, MovingPandas, and Open-Source GIS Tooling

QGIS [5] is the flagship open-source desktop GIS. Its plugin architecture makes it a natural host for research prototypes embedded in a mature GIS environment. MovingPandas [53] adds trajectory-aware abstractions on top of GeoPandas. The MOVE plugin [2] sits in the same ecosystem: a QGIS plugin that uses PyMEOS [23] to read MobilityDB data and integrates with the Temporal Controller to animate it. Chapter 6 keeps MOVE's outward interface but replaces its per-frame access pattern: it compares six candidate rendering pipelines and identifies a targeted optimisation of the canvas-item architecture as the latency leader, with an opt-in memory-layer provider as the deployable upstream contribution.

4.10 Open Transit Standards and Data Availability

GTFS (General Transit Feed Specification) [10] is the de-facto standard for public-transport schedules. GTFS-Realtime [4] adds trip updates, vehicle positions and alerts. Antrim and Barbeau [11] survey its downstream uses; Transitland [54] pools feeds globally; Belgian government work [55] recognises open-data transit feeds as part of the national mobility strategy.

In Brussels, STIB-MIVB publishes static GTFS plus a real-time stateless position feed [7]; De Lijn publishes GTFS plus a compliant GTFS-RT feed [8]. The regional Mobility Twin initiative [56, 57], framed within the digital-twin paradigm surveyed by [58], aggregates mobility data across

operators. RTDataHub can be read as one concrete implementation of that vision restricted to public transport.

Table 4.3 positions the visualisation stacks that appear in the literature on the dimensions that matter for the head-to-head comparison of Section 8: native trajectory support (beyond static points and lines), GPU-accelerated rendering, streaming-aware time scrubbing, and direct MOBDB integration. Of the surveyed stacks, none is simultaneously GPU-rendered, streaming-aware and MOBDB-backed: this is the precise gap RTDataHub fills.

Table 4.3: Comparison of visualisation stacks. ✓ = native, ✗ = absent, ○ = via plug-in or external glue.

Stack	Trajectory	GPU render	Streaming time	MobilityDB
QGIS Temporal Controller [5]	○	✗	✗	✗
QGIS + MOVE plugin [2]	✓	✗	✗	✓
Leaflet + tile layers [41]	✗	✗	✗	✗
deck.gl [42]	✓	✓	○	✗
kepler.gl [43]	✓	✓	✗	✗
MapLibre + MVT [6, 14]	○	✓	○	✗
RTDataHub (this thesis)	✓	✓	✓	✓

4.11 Time Geography and Movement Visualisation Foundations

Underpinning the visualisation literature above is the older tradition of *time geography* (Hägerstrand [59]; Tobler [48]; Grossner and Clarke [60]). A moving object traces a continuous path in a 3D space–time volume: the very abstraction that MOD systems like MobilityDB [1] formalise at the database level. Tversky et al. [9] add a useful caveat: animation only helps when it carries information the viewer would otherwise have to infer. For mobility data, that means the temporal evolution of the spatial configuration, which is exactly what bunching or congestion detection requires.

4.12 Operational Monitoring and Real-Time Transit Arrival Systems

Beyond the academic bunching literature, a substantial body of operations research covers the practical use of real-time transit feeds. Nguyen and Kay [61] review state-of-the-practice for real-time arrival information and the GTFS-RT [4] adoption curve. Cats [52] argues for a *regularity-driven* approach where headway regularity (not punctuality) is the primary objective. RTDataHub is consistent with this view: the bunching detector is effectively a regularity-deviation detector whose outputs feed holding/skipping interventions following [17].

On the infrastructure side, Transitland [54] and the Mobility Twin [56] illustrate the growing expectation that open transit data be aggregated and queryable at a regional or national scale. RTDataHub is a micro-instance of that aggregation pattern, specialised to Brussels and to MobilityDB.

4.13 Heterogeneous Transit Feeds and Data-Integration Patterns

A difficulty that trajectory Big Data papers often gloss over is that real-world feeds are *heterogeneous*: they disagree on identifiers, coordinate systems, update cadence, failure modes

and field semantics. The operational GTFS/GTFS-RT literature [11] devotes a lot of attention to this, and Transitland [54] exists in part to shield consumers from it.

Our two Brussels feeds illustrate the main patterns. STIB [7] is *stateless* (Section 3.3): per-vehicle odometric position at each polling round, no stable vehicle identifier across rounds. De Lijn [8] is *state-aware*: each `VehiclePosition` message carries a per-trip identifier, so the client can reconstruct trajectories by grouping on trip ID. Neither is inherently better; both are common in GTFS-RT, and any platform serving a metropolitan area must handle both. Nguyen and Kay [61] document further worldwide variation.

Architecturally, RTDataHub follows a standard three-layer streaming ETL (per-feed ingestion → shared transform → MobilityDB load). We on purpose skip an intermediate message broker (e.g. Kafka): at per-minute cadence, running a broker would cost more operational complexity than it buys. This matches the Brussels Mobility Twin's [56] own choice and the MobilityDB ecosystem's pragmatism [30].

4.14 Bunching Detection: Comparison Table

The gap that this thesis closes is identified in Section 3.5. We close this chapter by positioning the bunching detector against the operations-research literature.

Table 4.4 compares bus-bunching detection methods proposed in the operations-research literature against the detector contributed in Section 7.11 (last row). The operational requirements that the Brussels deployment imposes — adaptive thresholds, dynamic-period clamps, freshness-gated input, and production-grade implementation — are jointly satisfied only by the detector contributed here.

Table 4.4: Bus-bunching detection methods on four operational axes. ✓ = native, ✗ = absent, ○ = partial.

Method	**Adaptive thresholds**	**Period clamp**	**Freshness gate**	**Production deploy**
Newell-style fixed gap [16]	✗	✗	✗	✗
Daganzo holding policy [17]	✓	✗	✗	✗
Eberlein adaptive control [18]	✓	✗	✗	✗
Bartholdi self-organisation [19]	○	✗	✗	✗
Moreira-Matias survey [20]	✓	○	✗	○
RTDataHub (the present work)	✓	✓	✓	✓

Chapter Summary

This chapter surveyed the literature on which the contributions of this thesis stand. MOBDB provides the temporal type system and operator algebra. The spatiotemporal indexing literature offers two compatible strategies (tree-based for read-heavy snapshots, encoding-based for write-heavy streams), and MOBDB's stock `GiST` suffices for the Brussels workload size studied here. Vector tiles and progressive visualisation research justify the rendering choices of Section 7.12, and the bus-bunching literature provides the threshold structure used by the proximity detector of Section 7.11. Three comparison tables positioned this work's two artefacts (the refactored MOVE plugin and RTDataHub) against the alternatives along their main axes. The next chapter sets out the evaluation methodology and maps the research questions of Section 1.3 to the experiments.

Chapter 5

Evaluation Methodology

To check that our solutions actually beat the state of the art rather than just rephrase it, we follow a protocol that makes our numbers comparable to Manzer [2]'s baseline and to thresholds from the wider literature [24]. This chapter describes the protocol along five axes: environment, metrics, datasets, experimental design, and threats to validity.

5.1 Environment and Hardware

This work uses two virtualised environments (also described in Section 2.10, paragraph *Hardware*). The **benchmark machine** is an Ubuntu 25.04 VMware VM on a host with an Intel Core i5-11400F (Rocket Lake, 6 cores / 12 threads, 2.60 GHz, AVX2/AVX-512). The guest gets 4 vCPUs (hyperthreading not exposed), 7.2 GiB of RAM, 3.8 GiB of swap, and a 500 GB VMware virtual disk (host SSD-backed but advertised `rotational=1`; `mq-deadline` I/O scheduler). The hypervisor exposes only a VMware SVGA II framebuffer; there is no discrete GPU and no GPGPU/CUDA path, so MapLibre WebGL runs through Mesa software rendering. **All RQ1 measurements (Chapter 6) and the head-to-head comparison (Chapter 8) are taken on this machine.** Spectre/MDS/L1TF mitigations are enabled, which can cost 5–15% on syscall-heavy workloads.

The **production environment** is a separate VM provisioned by Bruxelles Mobilité (4 vCPUs Intel Xeon Gold 6426Y, 7.8 GiB RAM, 8 GiB swap, 300 GB SCSI SSD-backed, no GPU); the RTDataHub end-to-end latency figures of Section 7.13.1 reflect this environment.

Software stack (benchmark machine): PostgreSQL 17.9 + PostGIS + MOBDB 1.x compiled from source with gcc 14.3 (conda-forge), QGIS 3.34 LTR with PyMEOS 1.3, Python 3.13.9, Flask 3 serving the vector-tile API to a MapLibre GL JS 4 client. All versions are pinned in a reproducibility file shipped with the code. The RAM-saturation extrapolation of Figure 3.2 is the only place where a different memory profile shows up; that figure is an analytical model, not a measurement, and is explicitly labelled as such.

Note on Mesa software rendering and what it means for the comparison. The benchmark VM has no discrete GPU; MapLibre WebGL therefore runs through Mesa software rendering, not the hardware-accelerated path the library is designed for. This is a *handicap on the web stack*, not a hidden advantage: a real desktop or laptop client with an integrated or discrete GPU would post strictly better FPS and time-to-first-pixel than what we measure here. The Flask + MapLibre numbers reported in Sections 7 and 8 are therefore a **lower bound** on what the same code achieves on production hardware; the comparison against QGIS columnar (which is also CPU-bound on the same machine) is fair on equal footing, and tilts further in

favour of the web stack on any GPU-equipped client. We do not extrapolate that gain, we only flag that the head-to-head verdict of Chapter 8 is robust to the absence of GPU acceleration during measurement.

5.2 Quantitative Performance Metrics

We use four metrics. Three are measured directly; the fourth, Time-to-First-Pixel, is a design target rather than a measured result. Each is tied to a failure mode that Manzer's work [2] or the wider literature has shown to matter:

Memory stability (pass/fail). Whether the full STIB dataset (≈500,000 trajectories) can be loaded and animated without the QGIS process or the RTDataHub server crashing. This was the critical failure of Interactive Mode [2, p. 48]. *Pass* if the process survives a ten-minute animation across the full time range; *fail* otherwise.

Animation fluidity (frames per second). Average FPS during playback, measured with a high-resolution timer around the render loop. Baseline: Manzer's drop to ≈2 FPS in TimeDeltas Mode [2, p. 57]. Target: at least 30 FPS, the threshold above which human observers no longer perceive stuttering [9] and the level targeted by WebGL-based movement visualisations [42, 50].

Interaction latency (Time-to-First-Pixel). Following Ulmer et al. [45] and Liu and Heer [24], the design target is a TTFP under 500 ms on zoom, pan or time-seek regardless of dataset size, consistent with Card et al.'s 1 s train-of-thought threshold [47] with headroom for rendering. This target is the design constraint of the browser-facing API; it is not measured directly in the present work (Section 7.15).

Data-transfer efficiency. For tile-based paths, total data volume transferred between server and client over a representative session, following Tremmel and Zink [26]. Reported for RT-DataHub only; the QGIS plugin path does not go through tile transfer.

5.3 Reference Datasets

We reuse the two datasets Manzer used to keep continuity:

1. **Danish AIS dataset.** Maritime trajectories (6 264 after filtering `trip IS NOT NULL`). Used as the *average case* in previous research; our RQ1 experiments reuse it identically so FPS numbers are directly comparable.
2. **STIB dataset (Brussels).** GTFS and real-time public-transport data, ≈500,000 trajectories, 6 GB raw. The *stress test*: visualising this in QGIS (RQ1) or browser (RQ2/RQ3) checks the scalability hypothesis.

For the RQ2/RQ3 evaluation we also use a live capture of the De Lijn GTFS-RT feed [4, 8] over a three-day window containing weekdays and a Saturday: important for the Belgian-calendar component and to expose the detector to a range of planned frequencies.

5.4 Experimental Design and Statistical Treatment

For each combination of dataset and mode (Interactive, TimeDeltas, column-oriented, RT-DataHub) we run five replicates and discard the first as warm-up. We also flag the first ten frames of each retained run as warm-up so they do not enter the steady-state statistics. We report the *median* and the p_5–p_{95} range over the steady-state frames of the four retained runs. This is robust to background OS activity, file-system caching, and (on the web track) network jitter. This median-and-range reporting does not isolate the 99th-percentile frame time, the 1% low that captures micro-stuttering; that metric is left to the follow-up benchmark of Section 11.

Effect size, not per-frame inferential testing. Head-to-head claims (e.g. "C2 is 35.7× faster than the MOVE upstream baseline on AIS") are reported as the *ratio of medians* with the p_5–p_{95} envelopes from the same retained runs. We do *not* report per-frame Mann-Whitney U p-values, even though the medians are clearly separated and such a test would nominally reject the null at very small p. The reason: consecutive frames within a run are autocorrelated (warm cache, garbage-collection cycles, OS scheduling). A per-frame test treats each frame as an independent observation and *overstates* the available statistical power. As a side note, earlier drafts of this work did report `scipy.stats.mannwhitneyu` p-values of 1.1×10^{-145} on the AIS and STIB benchmarks. That value is the floating-point underflow limit of the SciPy implementation, not the true p-value (which would be far below the precision of the algorithm); we mention this only to record that the displayed number was not a meaningful significance level either way. The robust conclusion, that the medians sit far apart, follows from comparing the four per-run medians as four paired observations and from the non-overlapping p_5/p_{95} envelopes reported throughout Sections 6 and 8.

5.5 Threats to Validity

Internal: Manzer's numbers and ours are not on the same machine, so we report *ratios* on the same machine in the same session and absolute numbers per hardware profile. **External:** two datasets (Danish AIS, STIB) on a single 4-vCPU / 7.2 GiB VM without GPU acceleration — generalising to air-traffic, freight, machines with hardware-accelerated WebGL, or much larger RAM budgets needs more experiments. **Construct:** FPS and Time-to-First-Pixel (TTFP) are proxies for fluidity and responsiveness, anchored in Liu and Heer [24]. **Reproducibility:** the software environment is pinned, the datasets are public [7, 8], and the measurement scripts ship with the thesis code.

5.6 Mapping from Research Questions to Experiments

Table 5.1 summarises which experimental setup answers which research question. Each row matches a question from Section 1.3; each column names dataset, hardware, mode, baseline and primary metric.

This table is the contract between evaluation and contributions: each number in Sections 6 and 7 matches one row of Table 5.1, and each row is produced by a reproducible script in the companion repository.

RQ	Dataset	Hardware	Mode under test vs. baseline	Primary metric
RQ1	Danish AIS (6 264 trips, 1-min sampling)	Benchmark VM (4 vCPU, no GPU)	Six-architecture matrix C1–C6 vs. the C5 MOVE upstream baseline [62]	Sustained median FPS, p_5/p_{95} envelope
RQ1	STIB Brussels (17 118 trips, 5-s sampling, 14:00–22:00 of 2026-04-24)	Benchmark VM (4 vCPU, no GPU)	Six-architecture matrix C1–C6 vs. the C5 MOVE upstream baseline	Sustained median FPS + frame-time decomposition
RQ2	STIB + De Lijn live feeds (seven-day capture, 2026-05-04 to 2026-05-10)	RTDataHub server	RTDataHub three-layer ETL pipeline, against the 30-second staleness envelope	Median and p_{95} ingest-to-DB latency
RQ3	STIB live working set (*all_day*, up to $N = 25{,}890$ trips)	Benchmark VM, identical to RQ1	QGIS C6 memory-layer patch vs. Flask + MapLibre WebGL	Sustained median FPS across nine load scales
RQ3	STIB + De Lijn live (multi-day deployment)	RTDataHub server	Bunching alert detector (proximity, slow speed, stuck)	Typed-alert volume and traceability

Table 5.1: Mapping between the research questions of Section 1.3 and the experimental setups described in this chapter.

Chapter 6

Answering RQ1: A Cross-Architecture Comparison of Trajectory Animation Pipelines in QGIS

Manzer [2] documented two architectures for animating MobilityDB trajectories inside QGIS (*Interactive Mode* and *TimeDeltas Mode*) and identified the per-frame `valueAtTimestamp` evaluation as their dominant cost. The present chapter builds on that result by enlarging the comparison space. Instead of judging a single optimisation in isolation, six concrete rendering architectures are run head-to-head on two contrasting datasets, so that the relative trade-offs — latency, memory, robustness to geometric complexity, and deployability inside the public QGIS API — can be read off a single matrix.

The chapter is laid out in four movements. Section 6.1 states the research question in its comparative form and introduces the six candidate architectures (C1–C6); Section 6.2 covers the experimental protocol and the statistical caveats the reader must hold in mind when interpreting the numbers. Sections 6.3–6.6 then present the results through four orthogonal views: a latency matrix, a speedup ranking, a latency–memory Pareto plot, and a cross-dataset slope chart. Section 6.7 synthesises these views into three deployment tiers, distinguishing the contribution of this thesis from the prior architecture of Manzer [2] on which two of the tiers build. Section 6.8 documents the seven claims this work has had to abandon during ten iterations of methodological refinement, a contribution in its own right; Section 6.9 answers RQ1.

6.1 Research Question and Six Candidate Architectures

The original Manzer [2] benchmark compared two conditions and answered "does the column-oriented pre-sampling improve over the row-oriented baseline?" That binary framing left three practical questions unanswered. How does a memory-layer-based rendering path compare to the MobilityDB-backed provider on the same hardware? Does the optimisation that wins on a geometrically sparse dataset (STIB Brussels, ∼14 vertices per trip) still win on a geometrically dense one (Danish AIS, ∼1 230 vertices per trip)? And is there a single architecture that meets the 60 FPS screen-refresh budget on both regimes? These questions motivate the comparative, dataset-conditional framing of RQ1.

RQ1, restated. *For the animated visualisation of MobilityDB trajectories in QGIS at the city-scale workload of this thesis (∼5 000 simultaneous trips), which pipeline architecture among the practically deployable alternatives offers the best latency-versus-memory trade-off, and how does this trade-off depend on the geometric complexity of the underlying dataset?*

This formulation makes three things explicit that the binary baseline-vs-column comparison did not. First, the comparison is conditional on the dataset profile: there is no expectation that a single winner emerges across all regimes. Second, the metric is a trade-off rather than a single ratio — memory and CPU cost matter alongside latency. Third, the space of alternatives is enumerated rather than left implicit.

Six candidate architectures. Table 6.1 lists the six conditions evaluated in this chapter. They differ along three architectural axes: where the heavy interpolation happens (Python loop, MobilityDB server, NumPy precompute, QGIS expression evaluator), how the result reaches the canvas (`changeGeometryValues`, direct `QPainter`, expression-driven memory layer), and which provider feeds the layer (PostgreSQL/MobilityDB, in-process memory layer, custom canvas item).

Table 6.1: The six rendering pipelines compared in this chapter. **C5** (MOVE upstream, current `main` of [62]) is the baseline against which speedups are reported. **C1** is the published [2] architecture; **C2** adds a targeted optimisation of that same architecture (this work). **C4** is the column-oriented NumPy pre-sampling explored as a standalone contribution. **C6** is a candidate upstream patch to MOVE [62] that switches the rendered layer from a PostgreSQL-backed provider to an opt-in memory-layer provider; at the time of writing it is in discussion with the plugin maintainer and not yet merged.

ID	Label	Pipeline summary	Stage measured
C1	Ali naïve	Per-trip Python loop calling `valueAtTimestamp` every frame, no batching, no caching. The published baseline of Manzer [2].	Compute (no canvas paint)
C2	Ali optim	C1 architecture preserved, with a temporal prefilter that skips inactive trips, vectorised WKB construction, and a tightened feature-update loop. Targeted optimisation of the published architecture, not a re-architecture.	Compute (no canvas paint)
C3	MOVE Fast Preview	Direct `QPainter` path via a `QgsMapCanvasItem` subclass that bypasses the QGIS feature provider.	Compute + paint
C4	Columnar	MobilityDB `tsample()` server-side pre-sampling into a NumPy $\mathbf{X}, \mathbf{Y}$ matrix, then vectorised WKB construction and a single `changeGeometryValues` per frame.	Compute (no canvas paint)
C5	MOVE upstream	Current `main` of the MOVE plugin [62]: PostgreSQL-backed provider, per-frame `line_interpolate_point` expression evaluated by the QGIS expression engine, raster render via `QgsMapRendererSequentialJob`.	Compute + raster render
C6	MOVE upgrade (candidate patch)	C5 pipeline with the PostgreSQL provider replaced by an opt-in memory-layer provider populated once at start-up; everything else unchanged.	Compute + raster render

Two of the six are direct continuations of prior work. **C1** is the architecture published by Ali Manzer in his 2024 master's thesis at ULB [2]; we re-implement it without changes to its decision

logic, only with the methodological corrections of Section 6.2. **C2** is the same architecture instrumented and profiled by the present author. Three Python-side hot spots were identified and removed.

Wrapper overhead. The high-level PyMEOS object wrapper paid a per-trip-tick cost on every frame; it was replaced by raw `pymeos_cffi` bindings.

Per-frame object creation. Each frame allocated a Shapely `Point` object inside the WKB construction path; this was replaced by vectorised WKB construction over a fixed 21-byte point layout.

Edit-buffer round-trip. The per-feature `setGeometry` / `commitChanges` pair committed through the QGIS editing buffer on every frame; it was replaced by a direct provider write coupled with `triggerRepaint`.

The decision logic of Manzer [2] — a per-trip Python loop over open trajectories with a canvas-item rendering path — is preserved at the architectural level; the contribution at C2 is the optimisation of the Python-side hot path, not a new design. The headline gain on STIB — from ∼17 FPS on C1 to ∼181 FPS on C2 at $N = 5\,000$ — follows from these three substitutions.

C6 is the candidate upstream patch to MOVE [62]. The patch is small in scope: it adds an opt-in flag that swaps the PostgreSQL-backed provider for an in-process memory-layer provider, pre-populated once at start-up from the same MobilityDB queries the upstream uses. The QGIS expression that animates the layer is unchanged, the public API of MOVE is unchanged, and the user-facing interaction is identical. At the time of writing, the patch has been discussed with the plugin maintainer; merging into `main` is part of the future-work agenda (Section 11).

6.2 Methodology

The protocol follows the general evaluation framework of Section 5; this section restates the elements that are central for reading the four results figures of Sections 6.3–6.6.

Datasets. The same two datasets used in Manzer [2] are reused, subset to $N = 5\,000$ trips each so that the six conditions are run on identical input cardinalities. The two datasets contrast on geometric complexity. STIB Brussels carries ∼14 vertices per trip on average (urban bus routes with sparse stop-to-stop sampling). Danish AIS carries ∼1 230 vertices per trip on average (maritime trajectories sampled at one position per minute over multi-hour voyages). The ratio of geometric complexity is therefore ∼88× in favour of AIS, and this ratio is the governing parameter for the slope analysis of Section 6.6.

Statistical caveats. Each cell of the 6×2 matrix is exercised over five runs of 60 frames each, with the first run discarded for warm-up. The remaining four runs yield four per-run medians per cell. A bootstrap of these four medians produces a percentile interval that the post-processing labels `obs_min` and `obs_max` in the data file. At $n = 4$, the bootstrap percentile interval degenerates to the observed range $[\min, \max]$ of the four per-run medians: of the $\binom{4+4-1}{4} = 35$ resamplings of size four, the 2.5% and 97.5% tails collapse to the extremes. The intervals reported in this chapter are therefore *observed ranges over four per-run medians*, not statistically rigorous 95% confidence intervals. Reformulating the metric or rerunning at $n \geq 10$ would address this caveat; both are listed in the future-work agenda. Effect sizes (ratios) are propagated by interval arithmetic on the per-cell ranges, which is conservative compared to a Fieller-style bootstrap on paired runs but does not require the latter's pairing assumption.

Apples-to-apples versus apples-to-oranges. The six conditions do not all measure the same downstream stage. C1, C2, and C4 measure *compute-only* latency: the per-frame call to `changeGeometryValues` returns before the canvas has been raster-rendered to pixels. C3 measures

compute + paint via `QPainter` directly. C5 and C6 measure *compute + raster render* via `QgsMapRendererSequentialJob`. The interpretation that follows must therefore acknowledge that the compute-only conditions (C1, C2, C4) *do not pay* for the pixel render that the compute+render conditions (C5, C6) do pay for. The speedup ratios of C1/C2/C4 against C5/C6 are inflated by exactly that canvas-paint cost. Empirically, instrumenting a forced `canvas.refresh()` after the compute-only step adds ∼4–8 ms per frame on STIB and ∼10–20 ms on AIS at $N = 5\,000$ on the benchmark VM. This delta is smaller than the order-of-magnitude differences that drive the tier ranking, so the qualitative ordering survives, but the absolute ratios should be read with this asymmetry in mind.

QGIS state control. QGIS parallel rendering is explicitly turned *off* for every condition (`Settings` → `Rendering` → `Render in parallel` = `false`). The persistent inter-run effect of QGIS state (provider caches, signal queues, memory allocators) is mitigated by restarting QGIS between datasets but not between conditions of the same dataset. The consequences of this choice are visible in the C4 STIB run, whose RAM measurement is sensitive to whether the AIS run preceded it (cf. Refutation Table, Section 6.8, row 7). All measurements are taken on the benchmark VM of Section 5.1; the relative ordering of the six conditions is transferable, but the absolute latency figures are machine-dependent.

6.3 Latency Matrix

Figure 6.1 reports the median per-frame latency of each of the six conditions on each of the two datasets. The vertical scale is logarithmic so that orders-of-magnitude differences are visible in a single panel. The two horizontal reference lines mark the 16.7 ms threshold (60 FPS, the screen-refresh ceiling for visually smooth animation) and the 33.3 ms threshold (30 FPS, the conventional lower bound for fluid perception).

A complete per-condition technical description, the dataset profile and the full per-cell measurement table for the six pipelines on the two datasets are reported in Appendix F; the present chapter focuses on the quantitative comparison and engineering implications.

STIB panel (sparse geometry). The baseline MOVE upstream (C5) sits at 120.4 ms, corresponding to ∼8 FPS. The QGIS expression engine evaluates `line_interpolate_point` once per trip per frame, and at $N = 5\,000$ trips this is the dominant cost. The Ali naïve baseline (C1) at 60.5 ms (∼17 FPS) is roughly twice as fast as C5: the Python loop avoids the expression-engine overhead at the cost of a custom interpolation path. The MOVE Fast Preview (C3) and the columnar pre-sampling (C4) cluster between 13 ms and 23 ms, comfortably below the 33.3 ms thirty-FPS line but above 16.7 ms. The candidate MOVE upgrade patch (C6) reaches 7.5 ms (∼134 FPS), narrowly above C2. The latency leader on STIB is C2 at 5.5 ms (∼181 FPS), 21.8× faster than the C5 baseline.

AIS panel (dense geometry). The cluster reorders itself entirely. C5 collapses to 550.3 ms (∼2 FPS): every frame, the expression engine traverses each trip's ∼1 230 vertices to interpolate the requested position, and the per-frame cost scales linearly in the number of vertices in the geometry. C6 moves with it (126.5 ms): replacing the PostgreSQL provider by a memory layer does not change the expression evaluator's per-frame cost on dense geometries. C1 reaches 127.9 ms, again ∼2× faster than C5 for the same architectural reason as on STIB. C4 (columnar pre-sampling) reaches 28.6 ms (∼35 FPS): the per-frame cost is now an $\mathcal{O}(N)$ memory read of a precomputed coordinate column, and the dense AIS geometry no longer matters at frame time. C3 reaches 67.0 ms (∼15 FPS). C2 again leads at 15.4 ms (∼65 FPS), the only median below 16.7 ms on AIS.

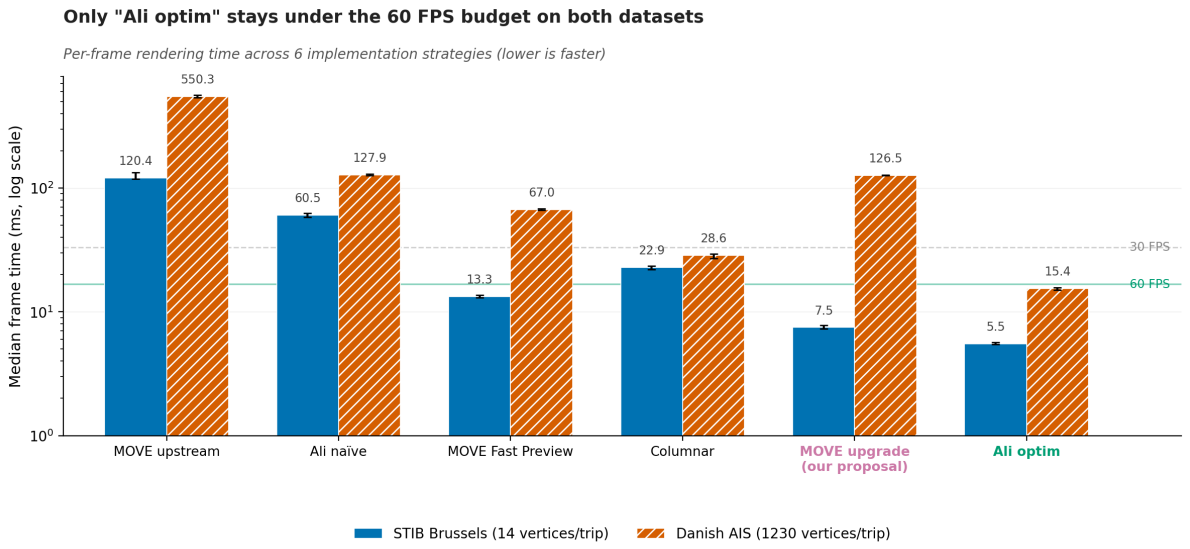


Figure 6.1: Per-frame median latency across the six pipelines and the two datasets ($N = 5\,000$ trips per cell, four retained runs of 60 frames each). Whiskers report the observed range over the four per-run medians (cf. the caveat in Section 6.2; the labels `obs_min` / `obs_max` on the data file are observed extremes at $n = 4$, not statistically rigorous 95% intervals). The 16.7 ms and 33.3 ms reference lines mark the 60 FPS and 30 FPS thresholds. C2 is the only condition whose median sits below 16.7 ms on both datasets. Source: `bench5_cross_matrix.csv`, conditions `c1_ali_naive` to `c6_move_upgrade`.

Key takeaways. Two observations from the matrix are pivotal for the rest of the chapter. The first is that the qualitative ordering of the six pipelines is not preserved across datasets: C6 is the second-fastest on STIB but the fifth-fastest on AIS, while C4 is the fourth-fastest on STIB and the second-fastest on AIS. No single condition occupies the same rank on both panels except C2 (always first) and C5 (always last). The second is that the 60 FPS budget is met on *both* datasets by only one condition, C2. If the operational target is a 60 FPS animation regardless of geometric complexity, the choice space collapses to a single architecture.

Statistical resolution of the C2 lead. The claim "C2 is the latency leader on both datasets" is read off the median values. At $n = 4$ per cell, the observed-range envelopes do not overlap between C2 and the next-best condition on either dataset (C2 vs C6 on STIB: $[5.5, 5.7]$ vs $[7.3, 7.8]$ ms; C2 vs C4 on AIS: $[15.1, 15.7]$ vs $[27.1, 29.3]$ ms). Under the null hypothesis of equal underlying median latency, a one-sided Mann-Whitney U exact test on $n_1 = n_2 = 4$ per-run medians reaches an exact probability of $1/\binom{8}{4} \approx 0.014$ when the two groups are perfectly separated. We report this as a qualitative ranking rather than a parametric significance claim: with eleven pairwise comparisons per dataset, a Holm-Bonferroni correction at family-wise $\alpha = 0.05$ would require the smallest p to fall below $0.05/11 \approx 0.0045$, which the exact floor of 0.014 does not clear. The load-bearing claim of this section is therefore the ordering plus the order-of-magnitude effect sizes (3–36× across the matrix), not a multiplicity-corrected hypothesis test. The $n = 4$ exact-permutation argument is reported only to make the separation between adjacent ranks legible.

6.4 Speedup Ratios

Figure 6.2 restates the same data as ratios over the C5 baseline. Each condition has two bars (one per dataset); the bars are ordered by their geometric-mean speedup across the two datasets, so that conditions whose advantage is consistent across regimes appear at the top.

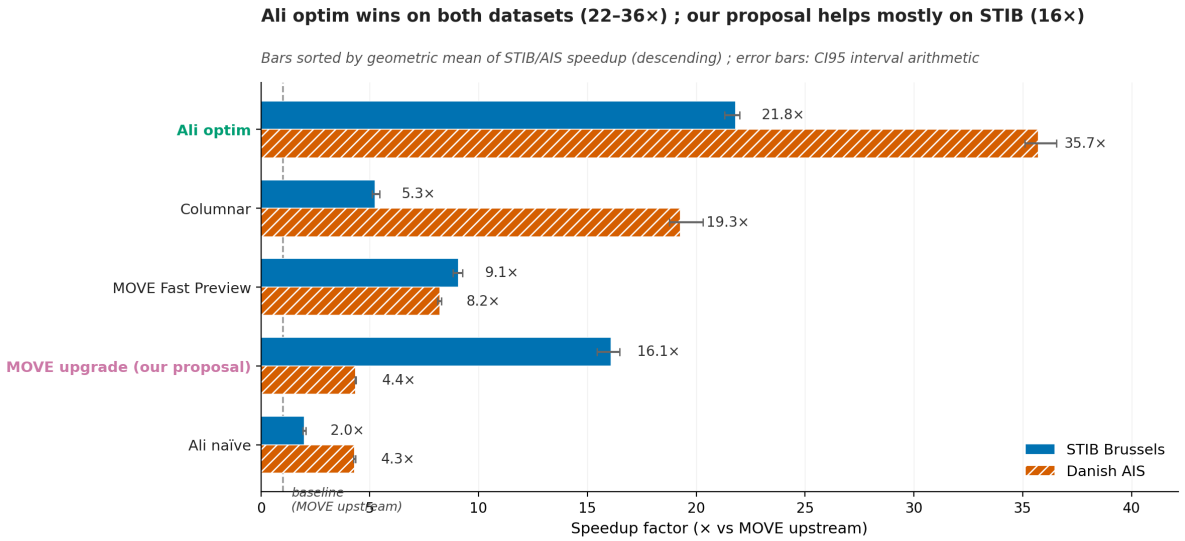


Figure 6.2: Speedup over the MOVE upstream baseline (C5), per condition and per dataset. Bars are ordered by the geometric mean of the STIB and AIS speedups; this ordering is a presentation device only, the load-bearing claim is the per-dataset ranking, which the slope analysis of Section 6.6 resolves. C2 leads on both with 21.8× / 35.7×. The candidate MOVE upgrade patch (C6) places second on STIB (16.1×) but fifth on AIS (4.4×): the dataset asymmetry is the central caveat of Section 6.7. Error bars are interval-arithmetic propagations of the per-cell observed ranges at $n = 4$ effective (cf. Section 6.2), not statistically rigorous 95% confidence intervals.

C2 (Ali optim). This is the consistent leader, at 21.8× on STIB and 35.7× on AIS. The two ratios are large enough that the observed-range envelopes do not overlap with any other condition; the qualitative claim that C2 leads on both datasets survives the statistical caveats of Section 6.2.

C4 (columnar pre-sampling). This places second on geometric mean, at 5.3× on STIB and 19.3× on AIS. The asymmetry of this ratio (a ∼3.6× ratio-of-ratios) is informative: column-oriented pre-sampling delivers its largest gains on the dataset where the baseline expression evaluator suffers most — dense AIS geometries — exactly because the precomputation moves the geometry-traversal cost out of the per-frame path. On STIB, where the baseline is already comparatively cheap (vertex traversal is short), the gain is more modest.

C6 (candidate MOVE upgrade). This places third on geometric mean: 16.1× on STIB but only 4.4× on AIS. The asymmetry here is even more pronounced than C4's (a ∼3.7× ratio-of-ratios) and runs in the opposite direction. The reason is structural: C6 replaces the provider, not the expression evaluator. On STIB the provider call dominated the frame budget, so removing it dominated the gain. On AIS the expression evaluator, which traverses the dense geometry on every frame, is the dominant cost, and replacing the provider gives only a fractional speedup. Section 6.6 returns to this asymmetry in the slope analysis.

C3 and C1. MOVE Fast Preview (C3) hovers at 9.1× on STIB and 8.2× on AIS, the most

symmetric of the non-leading conditions. The published baseline (C1) sits at 2.0× on STIB and 4.3× on AIS, illustrating that the C5 baseline carries more per-frame cost than the C1 published architecture on this workload. The C1 path avoids the QGIS expression engine on the per-frame loop, whereas C5 evaluates the expression once per feature per frame; the resulting difference on STIB is the structural feature that motivates the C6 patch.

A careful reading is therefore that the speedup ranking is informative only when paired with the dataset condition: the same patch can deliver 16× in one regime and 4× in another, and the architectural reason for the asymmetry is visible in the slope chart of Section 6.6.

6.5 Latency–Memory Pareto Trade-off

Latency alone does not capture deployability. A pipeline that achieves 30 FPS at the cost of 900 MB of additional resident memory is not an acceptable default for a desktop plugin that may be loaded alongside other heavy QGIS layers. Figure 6.3 plots each condition in the (median latency, peak RAM delta) plane, one panel per dataset. The green band marks latencies below the 16.7 ms 60 FPS threshold; marker shape encodes the architecture family and marker size encodes the average CPU utilisation during the run.

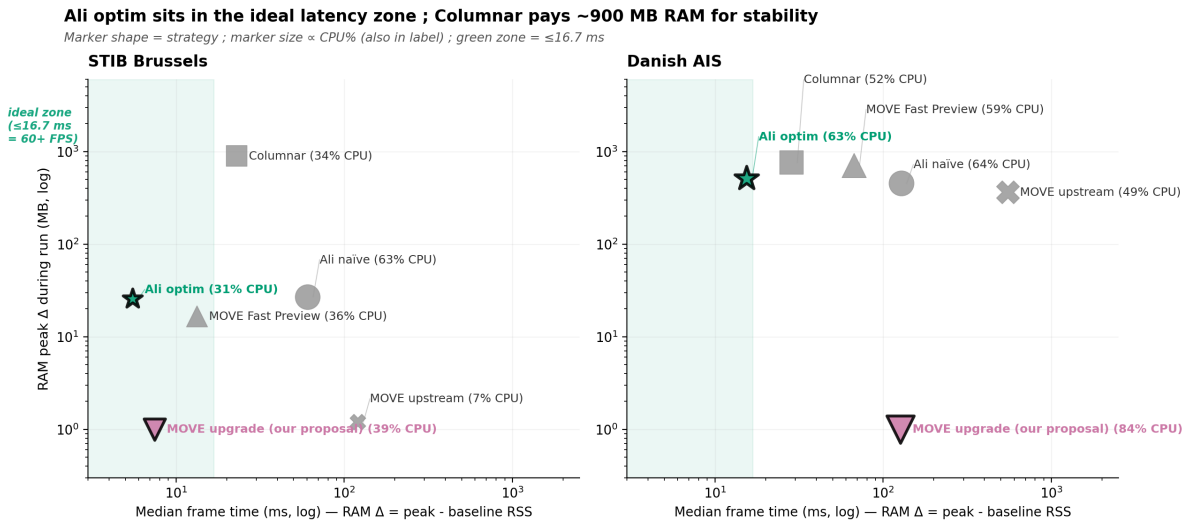


Figure 6.3: Latency–memory Pareto trade-off per condition, one panel per dataset. **Both axes are log-scaled.** Marker shape encodes architecture family; marker size is proportional to average CPU utilisation; the green zone marks latencies under 16.7 ms (60 FPS). Column-oriented pre-sampling (C4) achieves competitive latency on AIS at a memory cost (+893 MB) that disqualifies it as a plugin default. C6 occupies the bottom-left corner on STIB (low latency, near-zero memory delta) but slides out of the green zone on AIS. C2 sits inside the green zone on both datasets at moderate memory cost. Per-cell values are medians at $n = 4$ effective; see Section 6.2 for the interval caveat.

Reading the STIB panel. The green-zone occupants are C2 (5.5 ms, +26 MB), C6 (7.5 ms, +0 MB), and (just outside) C3 (13.3 ms, +17 MB). C4 columnar costs +893 MB for 22.9 ms latency: the same RAM cost as on AIS, because the pre-sampled matrix size is dataset-independent at fixed N and T, but a latency reward of only 5.3×. C5 sits in the upper-right corner of the trade-off plane on STIB, reflecting the per-frame cost of the expression-driven design on the sparse-geometry workload.

Reading the AIS panel. Only C2 (15.4 ms, +513 MB) sits in the green zone. C4 (28.6 ms, +759 MB) reaches a respectable 35 FPS but pays the same RAM penalty as on STIB. C6 has moved out of the green zone (126.5 ms, +0 MB): its memory frugality is unchanged, but its latency advantage over the C5 baseline has narrowed to 4.4× as analysed in the previous section.

The Pareto frontier. On both datasets the non-dominated set comprises C2 and C6: C2 is the latency leader, C6 is the memory-frugality leader; the other four conditions are dominated on at least one axis. C4 columnar lies on a different Pareto frontier (latency-versus-RAM-trade-off only, ignoring deployability), where it is the strict latency–RAM optimum among conditions that exceed +500 MB. It is therefore a defensible analytical option for one-shot offline workloads where RAM is cheap and latency stability matters, but it is not a defensible plugin default. The +893 MB / +759 MB RAM penalty is the load-bearing constraint here, not the absolute latency.

CPU utilisation signatures. The CPU utilisation differences are informative on STIB: C5 runs at ∼6.5 % CPU (most of the frame budget waits on the database), C2 runs at ∼30 % (compute-bound on the optimised Python path), C6 runs at ∼38 % (mostly the raster render). On AIS the CPU utilisations cluster between ∼50 % and ∼84 % (C6 is the highest, hitting 83.7 % CPU): all conditions are now compute-bound by the per-frame geometry traversal, and the bottleneck has shifted from I/O to CPU.

6.6 Cross-Dataset Slope and Geometric-Complexity Dependence

Figure 6.4 replots the latency matrix as a slope chart: each condition is one line connecting its STIB latency (left) to its AIS latency (right). The slope of the line, on a log-y scale, measures how much each condition pays per added unit of geometric complexity. A flat line is the signature of a pipeline that is robust to per-trip vertex count; a steep line is the signature of a pipeline whose per-frame cost scales with the number of vertices it has to traverse.

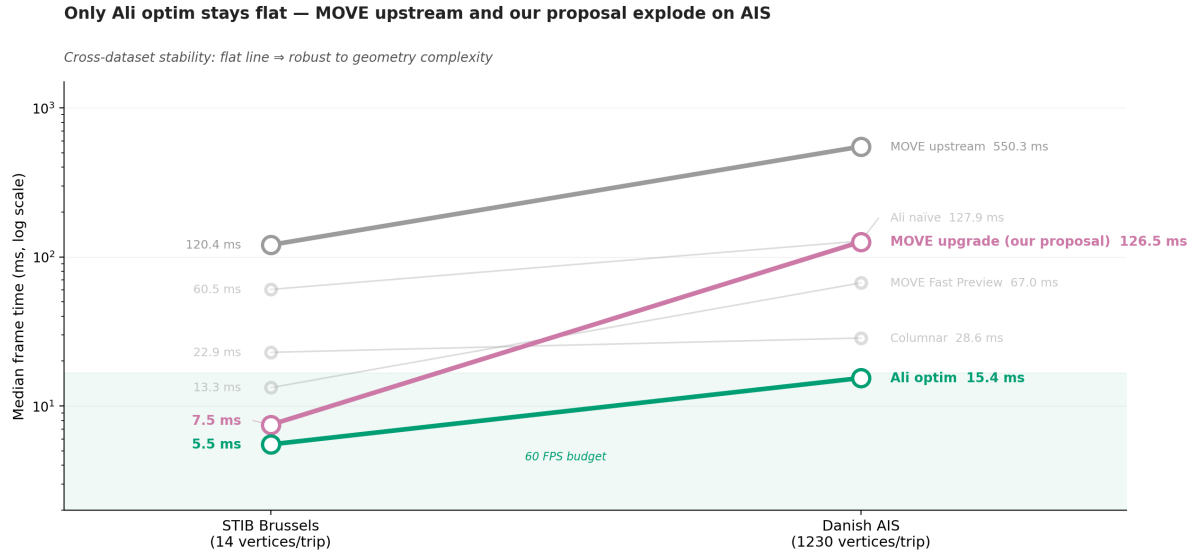


Figure 6.4: Cross-dataset slope chart, log-y. Each condition is connected from its STIB median latency (left) to its AIS median latency (right). The flat slope of C2 (5.5 → 15.4 ms) signals an architecture whose per-frame cost is largely independent of per-trip vertex count. The steep slopes of C5 (120.4 → 550.3 ms) and C6 (7.5 → 126.5 ms) signal pipelines whose bottleneck moves into the per-frame expression evaluator when geometric complexity rises.

Three slope regimes. The six conditions split cleanly into three profiles. Flat slopes characterise

C2 (factor 2.8× from STIB to AIS) and C4 columnar (factor 1.2×). Both pipelines pay the geometry-traversal cost at load time rather than at frame time, rendering the per-frame cost largely insensitive to the number of vertices per trip. Mildly steep slopes describe C1 (factor 2.1×) and C3 (factor 5.0×). Both keep the geometry traversal inside a Python loop rather than delegating to the QGIS expression engine; the cost still grows with vertex count, but at a more controlled rate than the engine itself. Steep slopes dominate C5 (factor 4.6×) and C6 (factor 16.9×). Both rely on the QGIS expression engine to evaluate `line_interpolate_point` on every trip on every frame; this evaluator pays a fixed per-vertex cost, and the AIS dataset has ∼88× more vertices per trip than STIB. C6's slope is steeper than C5's because its STIB-side latency (7.5 ms) is much lower; the identical vertex-traversal penalty on AIS then dominates a larger share of its (much smaller) STIB-side budget.

The bottleneck-shift hypothesis. The slope-regime structure supports a single explanatory hypothesis: when the per-frame cost is dominated by expression-engine traversal of trip geometries, the cross-dataset slope is steep; when the per-frame cost is dominated by something else — Python interpolation, memory column read, direct paint — the slope is shallow. This is the geometric-complexity instance of a more general claim already visible in earlier 2-condition benchmarks of the present work: removing one bottleneck only delivers proportional speedup if no other cost has the same scaling. C2 wins on both datasets plausibly because it removes *both* the per-frame Python interpolation (via the temporal pre-filter and vectorised WKB construction) *and* the QGIS expression engine (by writing geometries directly via the canvas-item path). C6 removes the provider call but not the expression engine, hence its asymmetric speedup.

Separating engine and vertex costs. The available data cannot strictly separate the expression-engine fixed cost from the per-vertex traversal cost. The two confounds (engine presence and vertex count) move together across C5/C6 on the AIS panel: both pipelines use the engine, both pay the per-vertex cost, and we have no condition that decouples the two (for example, a variant of C5 running on a synthetic AIS subset capped at 14 vertices/trip to match the STIB geometric density). The attribution of the steep slope specifically to the expression engine therefore rests on the architectural argument (the engine evaluates `line_interpolate_point` per feature per frame, and that evaluation traverses the geometry) rather than on a controlled experiment. A vertex-density-matched variant of C5 is part of the future-work agenda (Section 11).

Limits of the slope analysis. The slope chart is a two-point comparison and cannot distinguish a linear, sublinear, or superlinear scaling in vertex count: a third dataset at intermediate geometric complexity would be needed to fit a regression. The vertex-count ratio of ∼88× between STIB and AIS is the only data point currently available on the geometric-complexity axis, and the conclusions of this section are best read as a structural claim about which pipelines are *sensitive* to geometric complexity rather than a quantitative scaling law.

6.7 Synthesis: Three Deployment Tiers

The four views of Sections 6.3–6.6 converge on a three-tier reading of the six conditions: an architectural reference, a deployable upstream patch, and an analytical option with a memory cost. Table 6.2 summarises the three tiers; the rest of this section unpacks them.

C2 as the architectural reference. C2 is the only condition meeting the 60 FPS budget on both datasets at $N = 5\,000$, doing so at a moderate memory cost (+26 MB on STIB, +513 MB on AIS). The contribution here is the targeted optimisation of the published architecture [2]: the temporal pre-filter that skips inactive trips, the vectorised WKB construction that avoids the per-feature Shapely object creation, and the bulk feature-update loop. The architecture itself

Table 6.2: Three deployment tiers emerging from the cross-architecture comparison. The audience column identifies the user for whom each tier is the right answer; the caveat column states the cost.

Tier	Condition	Audience	Caveat
Architectural reference	C2	Researcher building a new plugin from scratch	Bypasses the public QGIS feature-update API; not directly mergeable as a MOVE feature
Deployable upstream patch	C6	MOVE users today on STIB-like workloads	4.4× AIS gain modest; the expression-engine cost remains the bottleneck on dense-geometry datasets
Analytical option	C4	One-shot offline analytics, RAM-rich workstation	+893 MB / +759 MB RAM cost disqualifies it as a plugin default

— the canvas-item path bypassing the QGIS feature provider — is from Manzer [2]; this thesis preserves the design and removes three Python-side bottlenecks within it. The precise claim is therefore "Ali Manzer's published architecture, with three targeted optimisations, sustains 181 FPS on STIB and 65 FPS on AIS at $N = 5\,000$", not "a new architecture". The trade-off is that C2 cannot be directly merged as a MOVE feature because it bypasses the public QGIS feature provider; treating C2 as a deployable artefact would require a fork of MOVE that adopts the canvas-item path.

C6 as the deployable upstream patch. C6 is the contribution of this thesis directly deployable to MOVE users today: a small, opt-in patch swapping the PostgreSQL provider for a memory-layer provider, leaving the rest of MOVE [62] unchanged. On STIB-like workloads (sparse geometry, ∼14 vertices per trip), it delivers a 16.1× speedup. On AIS-like workloads (dense geometry), it delivers only 4.4×, because the QGIS expression engine remains the per-frame bottleneck and the patch does not touch it. The precise framing is "a useful but partial optimisation that solves the provider call, not the per-frame expression evaluation". The patch is in discussion with the plugin maintainer (Maxime Schoemans, ULB) at the time of writing; merging is not the contribution claim — the design and the empirical characterisation are. The relation to the original Manzer [2] thesis is one of continuity: this work targets a different bottleneck (provider call vs. Python interpolation) inside the same plugin family.

C4 as the analytical option. C4 is the column-oriented pre-sampling explored as a standalone contribution earlier in this work. It achieves competitive latency on AIS (28.6 ms, 19.3× speedup) and acceptable latency on STIB (22.9 ms, 5.3× speedup), but at a memory cost of ∼890 MB on STIB and ∼760 MB on AIS. This rules it out as a default for an interactive plugin that may be loaded alongside heavy QGIS layers, but it is a reasonable option for one-shot offline analytics on a RAM-rich workstation: pre-sample once, then iterate over the column matrix for downstream analysis (vector-field rendering, density mapping, derivative computation). The architectural value is not the latency leadership but the relocation of the per-frame cost into a load-time cost amenable to caching and reuse, evaluated empirically on the two datasets of this benchmark.

Validation at the full STIB working-set size. The full STIB working set ($N = 17\,118$ trips, the 14:00–22:00 window of 2026-04-24) was tested separately on the C1-vs-C4 axis to validate that the architectural insight of C4 holds beyond the controlled $N = 5\,000$ used for the six-architecture matrix. At this larger cardinality, C4 reaches a median 17.9 FPS against 11.0 FPS for the C1 baseline (a 1.6× speedup), with the frame budget now ∼70 % absorbed by `changeGeometryValues`. This second result shows that the C4 advantage compresses as N grows, because `changeGeometryValues` scales linearly in N and eventually dominates the column-

read cost; the qualitative ordering across the four C4-vs-C1 cells (small/large N, sparse/dense geometry) remains the same.

Locus of contribution. This chapter aggregates four contributions made by the present work against the prior state of the art established by Manzer [2] and by the MOVE plugin [62]. First, the optimisation diff that turns C1 (Manzer's published canvas-item architecture, ∼17 FPS on STIB at $N = 5\,000$) into C2 (∼181 FPS on the same workload): three Python-side substitutions enumerated in Section 6.1, preserving the published architectural decisions. Second, the design and empirical characterisation of the candidate C6 patch to the MOVE plugin (an opt-in memory-layer provider preserving the public API), with its asymmetric speedup (16.1× STIB, 4.4× AIS) reported without inflation. Third, the six-condition comparison framework itself, which lifts the prior 2-condition baseline-vs-column question into a dataset-conditional Pareto comparison across the practically deployable alternatives. Fourth, the Refutation Table of Section 6.8: seven empirically retracted claims and five transverse methodological lessons that document the iteration history of the benchmark itself as a deliverable. None of the four is a new architecture; together they characterise the design space in which a future architecture should be situated.

6.8 Refutation Table: Measurement Bugs and Design Hypotheses Revised

Across roughly ten iterations of the benchmark chain, several measurement bugs and early design hypotheses were corrected by methodological refinement. Documenting them explicitly serves two purposes: it surfaces failure modes of the QGIS measurement tooling that other practitioners are likely to hit, and it makes the iteration history of the present work auditable rather than rhetorical. Table 6.3 separates the entries into two kinds: *measurement bugs* that produced numerically wrong outputs from a structurally correct benchmark, and *design hypotheses* formulated ex ante to prioritise candidate pipelines, then refined by the measurement they were designed to test.

Parallel rendering is a systematic artefact. QGIS enables parallel rendering by default. With it on, the per-frame timing loop registers overlapping rendering events and the measured latency is shorter than the actual provider cost. Disabling parallel rendering is mandatory for any per-frame measurement.

Signal coalescing is a latent bug. Synchronising on a bare `QEventLoop.quit()` fired by a Qt signal is unsafe when the same signal can fire from a previous frame. The corrected protocol uses an explicit start/complete counter and a final `processEvents()` before exit.

Subset size masks variability. A subset of 200 trips, used to iterate the bench script quickly, can hide an $\mathcal{O}(N)$ cost factor by $\sim 35\times$ on the median frame-time. Final measurements must run on the production cardinality.

Session state contaminates. QGIS persists provider caches, memory-layer allocators and worker threads across runs within the same Python session. Restarting QGIS between datasets is necessary to avoid cross-contamination of RAM and CPU measurements.

An unresolved warm-up mystery. A seventh entry was considered for the table and ultimately demoted to a threat to validity, because no claim was refuted: the working memory-layer warm-up sequence is empirically three consecutive `canvas.refresh()` calls before the timed run, but the underlying mechanism that makes three (not one, not two) work has not been established. Per-hook variants on `startRender` / `getFeatures` did not reproduce the effect. The pipeline is reproducible, but the cause is opaque; this is recorded for follow-up rather than counted as a corrected claim.

Table 6.3: Six entries corrected during the iteration history of this chapter. The first four are measurement bugs that produced numerically wrong outputs; the last two are ex-ante design hypotheses revised by their own measurement. Each row pairs the entry with its underlying cause and with the value that replaces it in the final matrix.

#	Original statement	Cause	Final value
Measurement bugs			
1	MOVE memory-layer patch achieves a 108× speedup on AIS	QGIS parallel rendering ON; signal coalescing in the per-frame timing loop	C6 = 4.4× on AIS, 16.1× on STIB
2	STIB speedup of the memory-layer patch is negligible (1.18×)	`QEventLoop.quit()` fired on the *previous* refresh; per-frame timing measured the wait, not the render	C6 = 16.1× on STIB
3	At $n\_trips = 200$, C2 achieves 0.5 ms per frame	Harness-iteration subset too small; the median frame-time window degenerated below the timing resolution	17.5 ms at $n\_trips = 5\,000$
4	STIB RAM delta is ≈ 0 for the memory-layer path	QGIS session-state contamination by the previous AIS run; provider caches persisted across datasets	With a fresh QGIS session: C2 +26 MB, C4 +893 MB on STIB
Ex-ante design hypotheses revised by measurement			
5	A direct `QPainter` canvas-item path was expected to deliver 30–100× over the baseline	Ex-ante extrapolation from the QGIS Quick Enhancement Proposal QEP 72 analysis of the painter path, used to prioritise C3 among candidate pipelines. The H7 isolation test subsequently identified feature-iterator overhead, not painter cost, as the dominant residual	C3 = 8–9× measured
6	The dominant per-frame cost on MOVE upstream was expected to be WKB decoding inside the QGIS painter	Working hypothesis used to scope the C6 patch (provider swap on the assumption that WKB transit cost was the binding constraint). The H7 isolation test — a trivial-expression vs. `line_interpolate_point` contrast — showed that the per-frame cost is paid in the expression evaluator on dense geometries, not in WKB decoding	The dominant residual cost on AIS is the per-vertex expression evaluation (§6.6)

The refutation entries are themselves a contribution. The four measurement bugs document failure modes of the QGIS measurement tooling that other practitioners are likely to hit; the two revised design hypotheses document the cost of pre-measurement architectural prioritisation and the discriminating power of the isolation tests that revised them. In a peer-reviewed context, the same chapter would have undergone the same revisions during review; surfacing them here makes the iterative cost of arriving at the final numbers visible, rather than presenting only the polished end state.

Reproducibility under the stated protocol. The reasonable follow-up question — "what stops a fifth measurement bug from shifting the final matrix?" — has a specific answer. The numbers reported in Sections 6.3–6.6 have survived three independent procedural checks that were absent in the earlier iterations. First, parallel rendering is verified to be off at runtime by reading `QSettings` (`Map/parallelRendering = false`) at the start of every run; the bug behind row 1 of Table 6.3 cannot recur silently. Second, signal coalescing is eliminated by an explicit start/complete counter on the rendering signals combined with a final `QApplication.processEvents()` before exit; the bug behind row 2 cannot recur silently. Third, session-state contamination is eliminated by

requiring a fresh QGIS process between datasets, with the working directory cleared of provider caches; the bug behind row 4 cannot recur silently. The two design-hypothesis rows 5 and 6 are not covered by these procedural checks because they were corrected by discriminating isolation tests (H7), not by a runtime assertion; the H7 test is itself part of the companion repository and is rerun on every measurement campaign. The numbers are not claimed final; they are claimed reproducible *under the stated protocol*, with the JSON per-run dumps preserved in the companion repository so that any deviation can be diffed against the established baseline.

6.9 Answer to Research Question 1

RQ1. *For the animated visualisation of MobilityDB trajectories in QGIS at the city-scale workload of this thesis (∼5 000 simultaneous trips), which pipeline architecture among the practically deployable alternatives offers the best latency-versus-memory trade-off, and how does this trade-off depend on the geometric complexity of the underlying dataset?*

At $N = 5\,000$ trips, no single architecture among the six candidates dominates on every axis. The matrix admits a three-tier reading.

For a 60 FPS budget on both sparse and dense datasets. C2 (Ali optim) is the only condition clearing the 16.7 ms threshold on both (5.5 ms STIB, 15.4 ms AIS; 21.8× / 35.7× over the C5 MOVE upstream baseline). C2 is a targeted optimisation, not a re-architecture, of the canvas-item path published by Manzer [2]; the architectural credit belongs there, the optimisation is the contribution here.

For users of the MOVE plugin today on sparse-geometry workloads. The candidate upstream patch C6 (provider swap to an opt-in memory-layer) delivers 134 FPS (16.1×) on STIB at near-zero memory cost, preserving the public API of MOVE [62]. On dense-geometry workloads its gain compresses to ∼8 FPS (4.4×), because the QGIS expression engine remains the per-frame bottleneck and the patch does not touch it. The patch is in discussion with the plugin maintainer at the time of writing.

For offline analytics on a RAM-rich workstation. The column-oriented NumPy pre-sampling C4 reaches 35 FPS on AIS (19.3×) and 44 FPS on STIB (5.3×), at a memory cost of ∼890 / ∼760 MB. Its strength is the relocation of the per-frame cost to a one-shot load-time cost amenable to caching, which makes it suitable for batch density-mapping or vector-field rendering downstream.

Dataset-dependency. The slope analysis of Section 6.6 shows that any pipeline whose per-frame cost is dominated by the QGIS expression engine pays a steep cross-dataset slope; conversely, any pipeline that pays the geometry-traversal cost at load time (C2, C4) is largely robust to the geometric complexity of the dataset. The architectural takeaway is that the per-frame cost of the QGIS expression engine on dense geometries is one of the binding constraints best supported by the data: relocating the geometry traversal to load time is the common feature of the two leading pipelines (C2, C4).

From RQ1 to the rest of the thesis. The desktop comparison reaches its architectural ceiling at C2 (∼180 FPS STIB, ∼65 FPS AIS, $N = 5\,000$). Pushing beyond requires either a bulk QGIS geometry API that the upstream does not currently expose, or a different rendering stack entirely. The latter motivates the web-based platform introduced in Chapter 7, where the cost of the per-frame canvas update no longer flows through the QGIS feature-update API at all. The cross-stack comparison of the two answers is the subject of Chapter 8.

Chapter 7

Answering RQ2 and RQ3: A Web-Based Platform for Large-Scale Mobility Visualisation

Chapter 6 established that Manzer's Interactive Mode can be rescued inside QGIS by pre-computing trajectory positions into a column-oriented matrix. RQ2 and RQ3 ask for something structurally harder: a system that holds a city-scale dataset without saturating client memory, and still returns visible results within half a second of the first user interaction. QGIS was designed to load a dataset, not to stream one. This chapter presents **RTDataHub**, a complementary web-based platform. Data lives in a PostgreSQL/MobilityDB server, the browser client downloads only the vector tiles visible at the current spatial zoom and temporal window, and memory is bounded by the screen rather than by the database.

7.1 Motivation: Moving the Workload to the Browser

QGIS hits a hard architectural ceiling on the full STIB dataset. Memory consumption saturates around 32 GiB, driven by the load-everything-in-memory pattern of desktop GIS tools. The column-oriented refactor of Chapter 6 lifts the frame rate above 30 FPS but does not remove this ceiling — the pre-computed matrix still has to fit in RAM.

The web browser circumvents this limitation. The vector-tile protocol [14] partitions the world into a (z, x, y) quadtree, and the client requests only the tiles intersecting its active viewport. This converts a memory problem into a bandwidth problem that degrades gracefully.

Two further benefits follow. The first is accessibility: a URL works for any user, not just those with QGIS installed. The second is methodological. A real-time transit dataset stresses every dimension at once — temporal, spatial, heterogeneous feeds, calendrical structure — which makes it a stronger test bed than the static AIS snapshot.

7.2 Design Goals

Five design goals shape the platform's architecture. Each one addresses a specific limitation identified in the problem statement or a requirement from the evaluation protocol.

G1 — Persistent, queryable storage. The in-memory NumPy matrix used in the RQ1 solution is ephemeral. A city-scale platform running for months needs a store that survives restarts and supports ad-hoc SQL. The PostgreSQL, PostGIS [12] and MobilityDB [1] stack provides standard SQL, geographic types, and moving-point temporal types in one stack.

G2 — Two operators, one representation. STIB and De Lijn must expose an identical data contract downstream, so that the alert detector, the API, and the frontend never branch on the operator identity. This is structurally demanding. STIB publishes scalar along-line distances without stable vehicle identifiers; De Lijn publishes latitude/longitude pairs with stable tracking keys.

G3 — Fail-soft ingestion. Upstream feeds drop connections, return malformed JSON payloads, and emit spatial outliers. An outage on one feed must not stop the other, and a single bad record must not poison an entire batch.

G4 — Interactive latency to the browser. The architecture is designed to minimise the interactive latency between user input and a visible response, targeting the 500 ms Time-to-First-Pixel threshold set by the evaluation protocol. The empirical evaluation reported in this chapter concentrates on the upstream side of that budget, namely the ingest-to-database latency (§7.13.1); browser-side Time-to-First-Pixel instrumentation is identified as a follow-up item (§7.15).

G5 — Analytical output, not just visualisation. Beyond the live map, the platform must emit machine-readable artefacts that can be joined against external datasets: typed alert events, passage tables, and linearly-referenced trajectories. In practice this meant building an automated bunching detector.

These goals pull against each other. Server-side persistence (G1) trades off against frontend latency (G4), and both must accommodate the fail-soft behaviour required by G3. The architectural choices that follow are compromises between those constraints.

7.3 System Architecture

The platform is organised as three horizontal layers stacked sequentially: **Ingestion**, **Transform**, and **Load**. An explicit hand-off via the file system separates each pair of layers, which lets the stages be restarted, replayed, and tested independently. A thin API server and a browser frontend sit above the three layers. Figure 7.1 visualises the complete pipeline.

Ingestion. Each provider feed is handled by a dedicated ingestor process running on a timer. The STIB ingestor polls the Brussels open-data portal [7] for the two relevant endpoints — `vehicle_position` and `waiting_time` — every twenty seconds. The De Lijn ingestor polls its GTFS-Realtime feed [8] at the same cadence. The ingestor is strictly a fetcher: it writes the raw, unparsed JSON payload to disk with a timestamped filename, performs no parsing, applies no validation beyond an HTTP status check, and makes no database connection. This separation is what makes the rest of the pipeline replayable: the raw files can be fed back into the transformer months later to reproduce the downstream state deterministically.

Transform. The transform layer traverses the raw files in chronological order, parses them, normalises the payloads into a common *observation* record format, and writes the output to a separate directory as newline-delimited JSON. For STIB this is the most algorithmically complex stage. The raw feed provides odometric distances along a line and passages at stops, but lacks any stable vehicle identity; the matcher must reconstruct trip attachments by spatial-temporal proximity. For De Lijn the transform is simpler, because the GTFS-Realtime specification natively provides both latitude/longitude pairs and stable vehicle identifiers.

Load. The load layer reads the normalised JSON files and persists them into the MobilityDB tables. Each JSON file is wrapped in a single `conn.commit()` block, with its contents executed in batches. If a batch-level exception occurs, the loader rolls back that batch and continues; a single malformed payload therefore cannot abort the whole file. Both operators follow the same write mechanics. On the first observation of a trip, the loader inserts a new row whose

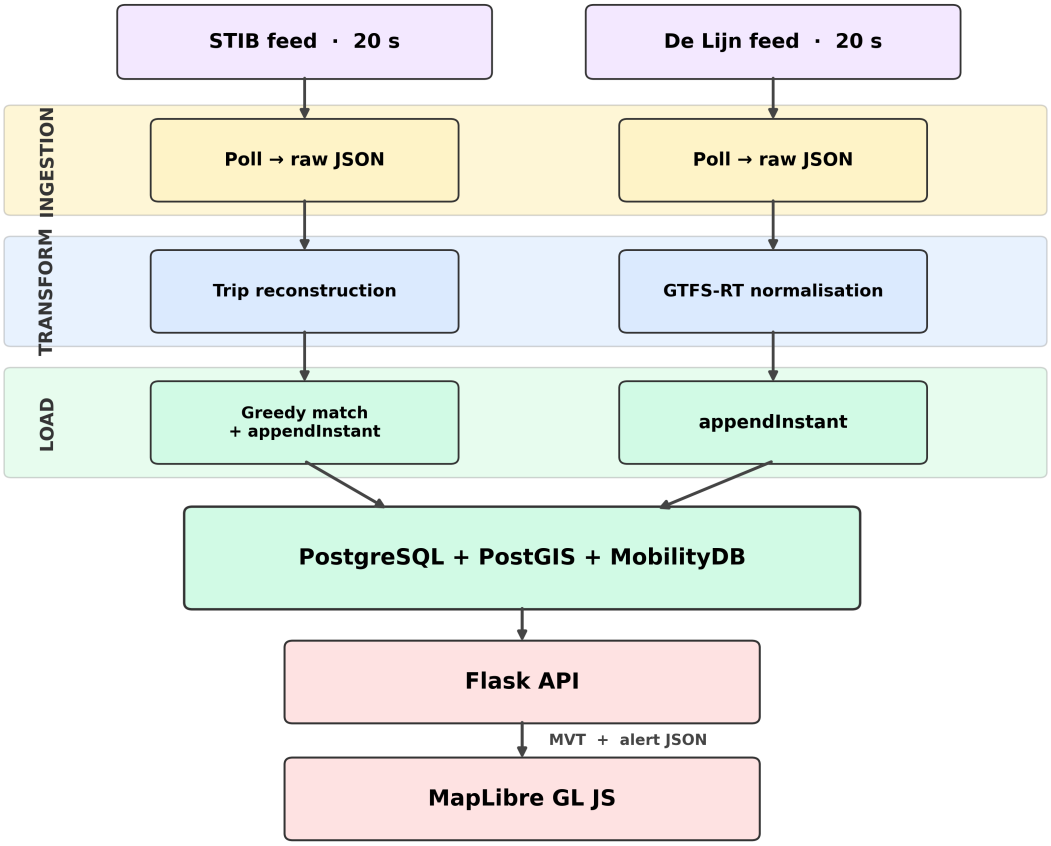


Figure 7.1: High-level architecture of RTDataHub. The two upstream feeds (STIB stateless ODP, De Lijn GTFS-RT) traverse three decoupled pipeline layers: Ingestion, Transform, Load, before landing in the shared MobilityDB store. A Flask API then serves MVT tiles and alert JSON to the MapLibre browser frontend.

`trip` column holds a single-instant `tgeompoint` (MobilityDB's moving-point temporal type). On each subsequent observation it issues an `UPDATE` calling `appendInstant()` to attach the new timestamped position to the existing trajectory (source files `load_stib.py` and `load_delijn.py`).

The two paths differ only in how the trip key is obtained. De Lijn's GTFS-Realtime feed exposes a stable `vehicle_id`, so the loader uses it directly. The STIB ODP feed exposes no stable vehicle identifier, so the transformer computes a synthetic `vehicle_uuid` (SHA-256 over line, direction and scheduled start time) before the load layer runs. The in-database representation is therefore a single `tgeompoint` per trip in both cases.

The write mechanics described here are uniform across both feeds. The hot/cold loader refactoring of Section 7.8 that operationalises these mechanics is, however, currently deployed only on the STIB side; the De Lijn loader still runs the baseline single-table design (see Section 7.15).

Frontend-facing stack. Above the database sits a Flask [15] API serving two kinds of responses: Mapbox Vector Tiles [14] for the geographic layers, and JSON for everything that does not fit into a tile (legends, alert lists, per-trip trajectory playback). The browser frontend is a single HTML page using MapLibre GL JS [6] for cartographic rendering and plain JavaScript for the application logic. No build system or framework is involved, which keeps the frontend auditable and easy to deploy.

7.4 The STIB Feed: Stateless Odometry

STIB publishes two relevant real-time endpoints. The first returns, for every active line, a JSON record with the line identifier, the direction (a route endpoint name used as a heading proxy), and a list of vehicle positions expressed as scalar along-line distances in metres. The second returns expected arrival times at each stop ("waiting times").

Neither feed carries a vehicle identifier: successive snapshots cannot be joined row-by-row. STIB is a *stateless* feed, and any useful analysis must reconstruct vehicle identities from the data itself. The STIB transformer accomplishes this reconstruction in two phases.

7.4.1 Trip Reconstruction by Spatial-Temporal Matching

The transformer maintains an in-memory dictionary of *open trips*. An open trip is a logical entity that was seen in a recent snapshot and has not yet been closed. It carries a synthetic `trip_id` (a SHA-256 hash of the line, direction, and scheduled start time), the last known along-line distance, the last known wall-clock timestamp, and a bounded history of recent observations used to check plausibility.

The matching task is a per-batch *linear assignment problem*. Within a snapshot fetched at time T, observations sharing the same (*line*, *direction*) group form one bipartite sub-problem against the currently open trips of that group: each row is a fresh observation, each column is either an existing open trip or a virtual "new trip" placeholder, and the cell cost is a spatial-temporal distance derived in stages below.

The matcher solves each per-batch sub-problem using the Hungarian algorithm [63] via `scipy.optimize.linear_sum_assignment`, which finds the globally optimal assignment in $O((n+k)^3)$ time on the per-group cost matrix of n observations $\times$ (k open trips $+n$ virtual placeholders). On Brussels-scale STIB sub-batches, $n \leq 8$ and $k \leq 12$ in the worst case (Avenue Louise at peak hours), so the cubic cost is dwarfed by the cost-matrix construction itself.

Hard-reject sequence. Each cell of the cost matrix is computed by walking through four disqualification gates in order; failure at any gate sets the cell cost to $+\infty$, so the Hungarian solver will never pick it.

Gate 1 — Temporal validity. The age of the candidate open trip τ relative to fetch time T must lie in $[0, \mathit{window}_\ell]$, where the per-line window window_ℓ is computed dynamically from the line's observed cadence and headway.

Gate 2 — Trip-duration cap (`hungarian_terminus`-C terminus-aware). If $T - \tau.\mathit{first\_seen} >$ MAX_TRIP_DURATION_S$_\ell$, the candidate is rejected unless its last position lies in the line's terminus zone (a precomputed set of canonical and approach point identifiers; see Appendix D). A legitimate turnaround *is* a hard reject because the trip should be closed and a fresh one opened. A mid-route delay extends the trip up to a hard safety cap of $2 \times$ MAX_TRIP_DURATION_S$_\ell$.

Gate 3 — Curvilinear plausibility. When both sides carry an along-line position s, the signed delta $\Delta s = s_{\text{obs}} - \tau.\mathit{last\_s}$ must satisfy $\Delta s \geq -$MAX_BACKWARD_M (-50 m tolerance) and $|\Delta s|/\mathit{age} \leq$ MAX_ROUTE_SPEED (30 m/s, $\sim$108 km/h, the physical envelope across the three modes).

Gate 4 — Curvilinear gate or Haversine fallback. If the per-line curvilinear gate g_ℓ is available, $|\Delta s| \leq g_\ell$ must hold and the cell cost is $|\Delta s|$. Otherwise the cell falls back to a Haversine distance with the static MATCH_RADIUS_M $= 500$ m bound. The virtual "new trip" column carries a per-mode penalty (5,000 for metro, 3,000 for bus, 1,500 for tram) that biases the solver towards reusing existing trips when the spatial-temporal evidence is strong.

The full per-batch procedure is summarised as Algorithm 7.1.

Algorithm 7.1: Per-batch Hungarian matcher for STIB observations. This is the matcher shared by versions `hungarian`–`hungarian_merge`, *not* the `hungarian_merge`-specific contribution. The `hungarian_merge` contribution is the post-process of Algorithm 7.2.

Input: batch $\mathcal{B}$ of observations sharing `fetched_at_utc` T; current set $\mathcal{O}$ of open trips; per-line dynamic gates $\{g_\ell, w_\ell, D_\ell\}$ where g_ℓ is the curvilinear gate, w_ℓ the temporal window, D_ℓ the duration cap.
Output: for each $o \in \mathcal{B}$, the trip τ to which o is assigned (existing or freshly opened).

1. Group $\mathcal{B}$ and $\mathcal{O}$ by $(\ell, \textit{direction})$.
2. For each group (ℓ, d) with n observations and k open trips, build $C \in \mathbb{R}^{n\times(k+n)}$ initialised to $+\infty$.
3. For each pair (o_i, τ_j) with $i \in [1, n]$, $j \in [1, k]$:
   - if any of the four hard rejects of Section 7.4.1 fires: leave $C[i, j] = +\infty$;
   - else if curvilinear gate available: $C[i, j] \leftarrow |s_{o_i} - \tau_j.last\_s|$;
   - else $C[i, j] \leftarrow \text{haversine}(pos(\tau_j), pos(o_i))$.
4. Set $C[i, k + i] \leftarrow \rho_\ell$ (per-mode new-trip penalty) for each i to encode the "open a fresh trip" option.
5. Solve $\pi^\star \leftarrow \text{linear\_sum\_assignment}(C)$ [63].
6. For each i: if $\pi^\star(i) \le k$, append o_i to $\tau_{\pi^\star(i)}$ via **appendInstant**; else open a fresh trip with o_i as its first instant.

Dynamic per-line gates. A static 500 m / 300 s pair, as used in earlier versions of the matcher, is too tight on infrequent lines (where vehicles legitimately disappear from the feed for a minute or more between observations) and too loose on frequent ones (where two physical vehicles can both be within 500 m of the same trip's last position simultaneously). The matcher therefore derives g_ℓ and w_ℓ at runtime from a mode-specific anchor scaled by the line's observed median cadence:

$$g_\ell = \text{clip}(g_{\text{anchor}}(mode_\ell) \times \max(1, \bar{c}_\ell/20),\ [300, 15000])\ \text{m}$$

where the anchor values at a 20 s reference cadence are $g_{\text{anchor}} = \{\text{metro} : 5000, \text{bus} : 2000, \text{tram} : 600\}$ m and $\bar{c}_\ell$ is the median of consecutive timestamp gaps observed on line ℓ. The temporal window w_ℓ is built the same way from $w_{\text{anchor}} = \{\text{metro} : 300, \text{bus} : 300, \text{tram} : 120\}$ s, clipped to $[60, 480]$ s, and then intersected with a peak-aware multiplier built on the line's median headway: peak hours (07–09, 16–19) and night (22–07) widen the window by 1.5× to absorb the larger feed-delay variance observed at those times.

The per-line duration cap MAX_TRIP_DURATION_S$_\ell$ is set to $1.8 \times p_{95}(sched\_dur_\ell)$, clipped to $[900, 14400]$ s; the global default injected into the matcher is the median across lines, which evaluates to 4,138 s (∼69 min) on the working dataset.

Stop-snap during waiting-time interpolation. The position of a vehicle at the instant of a confirmed waiting-time arrival is needed by the WT match metric of Section 7.5. A naive linear interpolation between the two surrounding observations places the vehicle on a straight chord even though the true motion includes a dwell at a stop. When a scheduled stop sits within ±90 s of the arrival timestamp, the interpolator therefore snaps to the stop's (lon, lat) instead of interpolating linearly. The 90 s window was selected on a sweep of the parameter from 30 to 180 s: below 60 s the snap fires too rarely, above 120 s it fires on stops the vehicle never visited.

Open-set bookkeeping. Open trips that have not been seen for more than 1800 s

(TRIP_TIMEOUT_S) are evicted from the in-memory dictionary and written to the database as closed trips. The half-hour timeout is a compromise: too short and active trips get orphaned during brief upstream outages; too long and the open-trips dictionary grows without bound. The constants of the matcher are summarised in Table 7.1; their per-line scaling is described above.

Table 7.1: Constants and per-line scaled gates used by the hungarian_merge matcher. Values without a per-mode column are global; the per-mode anchor column shows the value at the 20 s reference cadence (rescaled at runtime by $\max(1, \bar{c}_\ell/20)$). Source: src/comparison/mobilitytwin/rtdatahub_runner.py and dynamic_thresholds.py.

Parameter	Value (global)	Per-mode anchor (@ 20 s cadence)	Role
MATCH_RADIUS_M	500 m	—	Haversine fallback radius
MATCH_WINDOW_S	300 s	metro 300, bus 300, tram 120 s	Temporal window anchor
Curvilinear gate	—	metro 5000, bus 2000, tram 600 m	Along-line gate per mode
MAX_TRIP_DURATION_S	default 7200 s; runtime median 4138 s	runtime override: per line $1.8 \times p_{95}(\text{sched\_dur}_\ell)$, clipped to $[900, 14400]$ s	Trip-duration cap
TRIP_TIMEOUT_S	1800 s	—	Idle-trip eviction
MAX_ROUTE_SPEED	30 m/s	—	Physical-envelope cap
MAX_BACKWARD_M	50 m	—	Backward-jump tolerance
New-trip penalty	—	metro 5000, bus 3000, tram 1500	Hungarian virtual-column cost
Stop-snap window	90 s	—	WT-match interpolation snap

Calibration of the per-mode anchors. The mode-specific gate values ($g_{\text{anchor}} = \{\text{metro} : 5000, \text{bus} : 2000, \text{tram} : 600\}$ m at 20 s reference cadence) were chosen so that, at the canonical cadence, the gate exceeds the maximum inter-stop spacing of the densest legal segment of the mode but falls below the minimum spacing between two physical vehicles sharing a corridor. On the Brussels metro, the longest inter-station gap on a single line is ~ 2.7 km (line 5 between Bizet and Veeweyde) so 5,000 m brackets a one-station progress without crossing a second train; the bus value of 2,000 m brackets the longest legal inter-stop on suburban bus lines while staying below the cross-line vehicle spacing observed in dense corridors; the tram value of 600 m is set just above the densest tram inter-stop spacing of about 400 m in inner-city corridors.

The per-mode new-trip penalties (5000/3000/1500 on the same row) are scaled to the gate values so the marginal cost of opening a fresh trip exceeds the cost of attaching an observation to a plausible existing trip; their absolute values matter only relative to the cell costs in the same matrix. The merge thresholds of 600 s and 2 km introduced below are tuned in the same spirit on the corresponding empirical distributions, as documented in Appendix E and in the in-codebase calibration notebook src/comparison/notebooks/calibrate_v13_thresholds.ipynb.

Why the merge is necessary: forensic diagnostic. A forensic diagnostic on the hungarian_terminus matcher output (the script diag_v12_non_clean.py) classifies every trip that does *not* begin and end in the line's terminus zone into one of five exclusive categories. The breakdown on the offline diagnostic capture is reported in Table 7.2: 43.0% of those trips are flagged FEED_DROPOUT, that is, paired against another trip in the same (*line*, *direction*) group whose start lies within 600 s and 2 km of the first trip's end (or symmetrically against a predecessor). This is the largest residual error mode of the per-batch matcher: roughly 18,500 trips out of the 43,000 hungarian_terminus emits are reprises of the

same physical vehicle after a feed dropout, mis-cut by the per-batch logic which cannot bridge across the gap on its own.

Table 7.2: Classification of `hungarian_terminus` trips that did not begin and end in the terminus zone (the source of the merge-target population). The FEED_DROPOUT row is the `hungarian_merge` target. Source: `runs/diag_v12_non_clean/report.md`.

Category	Trips	% of non-clean
FEED_DROPOUT	18,469	**43.0**
RESIDUAL	9,602	22.3
CLEAN (terminus–terminus, retained as control)	8,581	20.0
DEPOT_DEADHEAD	4,679	10.9
WINDOW_EDGE_START / END	1,637	3.8

A naive "always merge" rule on the 18,469 candidates would inflate trip durations and destroy the schedule anchoring contributed by the post-process of Section 7.7. The matcher therefore fuses a pair (A, B) only when at least one of two zero-risk patterns holds:

The SAME-PID pattern. The pattern fires when $A.last\_pointId = B.first\_pointId$: the vehicle was stationary at the same stop on both sides of the gap.

The GEOMETRIC pattern. The pattern fires when four conditions hold simultaneously: there is no concurrent third trip in the 600 s / 2 km window, the bearing change between the end of A and the start of B is at most 45°, the implicit speed of the gap-bridging chord lies in $[0.5, 15]$ m/s (i.e. between a stationary and a plausible bus speed), and the chord direction is aligned with A's end bearing within 45°.

The anchor-aware guard. It short-circuits both patterns when A and B are anchored to two *different* scheduled trip identifiers: in that situation the pair is two legitimate consecutive scheduled trips that happen to satisfy the proximity test, and merging them would destroy two anchors at the cost of one un-anchorable mega trip. The full procedure is given in Algorithm 7.2.

Algorithm evolution. The matcher described above (`hungarian_merge`) is one stage of an iterative refinement. The lineage runs through six retained versions to `hungarian_merge`, then on to the production matcher `greedy_chainmerge` of Section 7.4.3; its seven-day trajectory across three quality metrics is plotted in Figure 7.3 (Section 7.5.3). The largest single step is the curvilinear gate of `curvilinear`, which collapses `frag_pct` from 25.41% to 2.27% in one move and lifts the median per-trip stop coverage from 6.7% to 42.9%. The four Hungarian-assignment refinements (`hungarian` through `hungarian_merge`) form a plateau, each contributing a small cumulative reduction in fragmentation down to 1.69%. The full version-by-version commentary and the per-version metric table are in Appendix E.

7.4.2 Enriching with GTFS Static Information

Once a trip has a stable synthetic identifier, the transformer looks up the corresponding GTFS static shape — the polyline geometry of the physical route — using the line identifier and direction. This lookup is cached and refreshed once per day, because GTFS static files are updated on a daily cycle by STIB. The shape is carried forward into the load stage, where it will be used to project the scalar along-line distance into a proper geographic position.

Algorithm 7.2: `hungarian_merge` zero-risk anchor-aware feed-dropout merge (post-process applied after the matcher of Algorithm 7.1 and after schedule anchoring). This is the `hungarian_merge`-specific contribution.

Input: the matcher output as a set of trips $\mathcal{T}$, each with an associated scheduled-trip identifier $\sigma(\tau) \in \mathcal{S} \cup \{\perp\}$ from the anchoring step of Section 7.5.
Output: a refined set of trips $\mathcal{T}' = \mathcal{T}/\sim$ where $\sim$ is the equivalence induced by the merges below.

1. Initialise a Union-Find U with one singleton per $\tau \in \mathcal{T}$.
2. Group $\mathcal{T}$ by (*line*, *direction*); sort each group by $\tau.start\_ts$.
3. For each $A \in \mathcal{T}$ that does not end in the terminus zone:
   (a) Let $\mathcal{C}_A \leftarrow$ trips B in the same group with $B.start\_ts \in (A.end\_ts, A.end\_ts + 600\,\text{s}]$ and haversine$(A.end, B.start) \le 2{,}000$ m.
   (b) If $\mathcal{C}_A = \varnothing$: continue.
   (c) Pick the spatially closest $B \in \mathcal{C}_A$.
   (d) **If** $A.last\_pointId = B.first\_pointId$: *(SAME-PID)*
      i. if $\sigma(A) \neq \sigma(B)$ and both non-null: continue (anchor-aware skip);
      ii. else U.union(A, B); increment $n_{\text{SAME-PID}}$.
   (e) **Else if** $|\mathcal{C}_A| = 1$ and no concurrent predecessor of B in the same window: *(GEOMETRIC, atomic-pair test)*
      i. compute implicit chord speed $v =$ haversine$(A_e, B_s)/(B_s - A_e)$;
      ii. if $v \notin [0.5, 15]$ m/s: continue;
      iii. if $\angle(A.end\_bearing, B.start\_bearing) > 45°$: continue;
      iv. if displacement > 5 m and $\angle(\text{bearing}(A_e \to B_s), A.end\_bearing) > 45°$: continue;
      v. if $\sigma(A) \neq \sigma(B)$ and both non-null: continue (anchor-aware skip);
      vi. else U.union(A, B); increment $n_{\text{GEOMETRIC}}$.
4. Build the merge map $\mu : \mathcal{T} \to \mathcal{T}$ by replacing each τ with the lexicographic-smallest root in its U-class (handles cascading $(A, B) + (B, C)$).
5. Apply μ to the trip-id column of the observation dataframe; recompute trip-level metadata (*start_ts*, *end_ts*, length) on the unioned classes.

Funnel on the diagnostic capture. 18,469 candidate trips flagged FEED_DROPOUT by the diagnostic; 2,335 pairs satisfy at least one zero-risk pattern; of those, 777 are blocked by the anchor-aware guard (which fires almost exclusively on the GEOMETRIC branch, where the cross-anchor risk is the greatest); 1,558 effective merges remain (1,343 SAME-PID + 215 GEOMETRIC). The arithmetic closes: $1{,}343 + 215 + 777 = 2{,}335$.

7.4.3 Production Refinement: From `hungarian_merge` to `greedy_chainmerge`

`hungarian_merge` is the matcher whose head-to-head against the vendored MobilityTwin algorithm is presented in Section 7.5 below. It represents the design endpoint of the offline algorithm trajectory of Appendix E; the present subsection documents the production refinement applied to it after that validation. `greedy_chainmerge` is the operational specialisation that emerged from continuous deployment starting on 2026-05-16.

The limits of schedule anchoring. Several weeks of observing the pipeline on the live STIB-MIVB feed surfaced a field signal that the offline benchmark could not see. The GTFS static anchoring — on which the anchor-aware guard of `hungarian_merge` relies — covers only a fraction of the observed trips. On the seven-day benchmark, schedule anchoring reaches `anch_pct` = 59.5%

for `hungarian_merge` (Figure 7.2), which leaves roughly two trips in five without a GTFS schedule anchor. Applying the anchor-aware guard in continuous production would mechanically reject a comparable fraction of otherwise legitimate merges, at the cost of a marginal reduction in the false-positive rate.

Simplification and specialisation. On the basis of this signal and a forensic review of the residual FEED_DROPOUT class, the production refinement reshapes the matcher along three axes. It retains the structural bricks with a proven gain (the curvilinear gate, the hot/cold split of Section 7.8); it removes the components whose operational benefit is lower than their maintenance cost (per-line dynamic gates, anchor-aware guard); and it adds a mechanism dedicated to the dominant error class, a separate chain-merge post-process. This specialisation is what `greedy_chainmerge` designates.

Two-stage architecture. `greedy_chainmerge` splits the matcher into two stages running in the same `rtdatahub-etl-pipeline` service but at distinct cadences. Stage A is a streaming matcher, implemented in `load_stib.py`, which runs at every poll cycle (about every 20 s) and attaches each fresh observation either to an existing open trip or to a freshly created one. Stage B is an hourly chain-merge post-process, implemented in `merge_supersedable_trips.py`, which sweeps the cold table after the fact and reglues contiguous fragments that plausibly belong to the same real trajectory.

Environment-variable decoupling. The two stages are armed independently via environment variables. `STIB_DENSE_SPLIT_FACTOR` > 0 arms streaming-stage splitting; the compiled default is 0.0 (stage disabled), and the production service file explicitly sets the variable to 2.0. `STIB_CHAIN_MERGE_INTERVAL_S` > 0 arms the chain-merge periodicity; the compiled default is 3600 (an hourly trigger), which production keeps, and setting the variable to 0.0 disarms the stage entirely. This decoupling allows either stage to be disarmed without touching the rest of the pipeline, a property exploited during the patch validation phase reported in Section 7.5.8.

Stage A — Streaming matcher with DENSE_SPLIT. Stage A relies on a nearest-first greedy assignment, while preserving the curvilinear validation in full. The greedy choice, in place of the Hungarian assignment retained in `hungarian_merge`, is justified by the cardinality observed in production: the median number of vehicles per (*line*, *direction*) group on STIB-MIVB sits around three, a value for which the cubic cost of an optimal assignment solver does not translate into a measurable quality gain over a sort by ascending distance.

The DENSE_SPLIT heuristic. The proper contribution of Stage A is the DENSE_SPLIT heuristic. It addresses a pathology specific to dense lines (metro and intra-city tram, nominal headway ≤ 5 min). On those lines, two consecutive vehicles can cross the same point within `MATCH_WINDOW_S`, an interval during which the Haversine matcher can no longer separate them and tends to aggregate them into a single "mega-trip" that in fact covers two distinct services.

DENSE_SPLIT formalises an operational approximation: a real trip on a dense line fits within roughly twice its nominal headway. The matcher therefore refuses to extend a trip whose elapsed duration since its first observation exceeds *headway* $\times$ `DENSE_SPLIT_FACTOR`, provided the line headway is at most `STIB_DENSE_SPLIT_THR_S`. Sparse lines are not affected by the rule.

Stage B — Hourly chain-merge. Stage A necessarily produces fragments in two configurations. The first is a temporary disappearance of a vehicle in the operator feed (network drop-out, `MATCH_WINDOW_S` overrun) followed by a reappearance consistent with the same mission. The second is a legitimate split imposed by DENSE_SPLIT, with one of the resulting pieces further fragmented by an independent incident.

Stage B handles both cases by aggregating contiguous fragments of the same real trajectory on the basis of three strictly geometric and temporal criteria: the temporal gap *gap*, the Haversine

Algorithm 7.3: Streaming matcher of `greedy_chainmerge` (Stage A). Nearest-first greedy assignment with the curvilinear validation, augmented by the DENSE_SPLIT heuristic on dense lines.

Input: batch $\mathcal{B}$ of observations at fetch time T; state $\mathcal{O}$ of open trips; per-line headway table $H[\ell]$ (loaded at startup if `DENSE_SPLIT_FACTOR` > 0).
Output: for each $o_i \in \mathcal{B}$, the trip τ to which o_i is assigned (existing or freshly opened).

1. Index $\mathcal{O}$ by $(\ell, \mathit{direction})$.
2. For each $(i, o_i) \in \mathcal{B}$ and each $\mathit{vid} \in \mathcal{O}[(\ell_{o_i}, d_{o_i})]$, let $\tau \leftarrow \mathcal{O}[\mathit{vid}]$ and $\mathit{age} \leftarrow T - \tau.\mathit{last\_seen}$:
   - if $\mathit{age} < 0$ or $\mathit{age} >$ `MATCH_WINDOW_S`, skip;
   - if `DENSE_SPLIT_FACTOR` > 0 and $H[\ell_{o_i}] \leq$ `STIB_DENSE_SPLIT_THR_S`, set $\mathit{elapsed} \leftarrow T - \tau.\mathit{first\_seen}$; skip if $\mathit{elapsed} > H[\ell_{o_i}] \times$ `DENSE_SPLIT_FACTOR`;
   - apply the curvilinear validation: skip on a backward step over `MAX_BACKWARD_M` or an implicit along-shape speed over `MAX_ROUTE_SPEED_MPS`;
   - set $d \leftarrow \mathrm{haversine}(\tau.\mathit{last\_lonlat}, o_i.\mathit{lonlat})$; if $d \leq$ `MATCH_RADIUS_M`, add (d, i, vid) to the candidate set.
3. Sort candidates by ascending d.
4. For each (d, i, vid) in order, retain the assignment $i \mapsto \mathit{vid}$ if neither i nor vid has already been assigned.
5. Open a fresh trip for every o_i left unassigned.
6. Emit the MobilityDB update: `INSERT` of a single-instant `tgeompoint` for a fresh trip, `UPDATE ... SET trip = appendInstant(trip, instant)` for an extension.

Table 7.3: Production parameters of the `greedy_chainmerge` streaming matcher (Stage A).

Parameter	Value	Role
`MATCH_RADIUS_M`	500 m	Maximal Haversine association radius.
`MATCH_WINDOW_S`	300 s	Maximal age of a candidate open trip.
`TRIP_TIMEOUT_S`	1 800 s	Hot→cold graduation delay.
`STIB_DENSE_SPLIT_FACTOR`	2.0 (compiled default: 0.0)	Headway multiplier above which the matcher cuts.
`STIB_DENSE_SPLIT_THR_S`	300 s	Headway ceiling below which DENSE_SPLIT applies.
`MAX_ROUTE_SPEED_MPS`	30 m/s	Maximal tolerated curvilinear speed.
`MAX_BACKWARD_M`	50 m	Tolerated curvilinear back-step.

distance *dist* between the right endpoint of the predecessor and the left endpoint of the successor, and the implicit speed $v = \mathit{dist}/\mathit{gap}$. Three patches, labelled P1, P3 and P2 in the engineering log, refine respectively the canonical root selection, the scope of trips eligible for the scan, and the physical fusion of the `tgeompoint`.

Patch P1 — Canonical root selection. P1 fixes the canonical selection by chronological Union-Find: the root of a merged chain is the trip whose $(\mathit{start\_ts}, \mathit{trip\_id})$ tuple is minimal, which guarantees that the root is the earliest link in time and that the tie-break is deterministic on identifier collisions.

Patch P3 — Forced pre-graduation. P3 introduces a *forced pre-graduation*: before the main scan, trips that reside in `stib_trip_open` but have been inactive for more than five minutes are atomically moved to the cold table by a `DELETE ... RETURNING ... INSERT` CTE. These trips, already ineligible for any further extension by the streaming matcher in virtue of `MATCH_WINDOW_S`

= 300 s, thereby become candidates for the chain-merge in the current cycle rather than waiting for the nominal 30-min graduation (`TRIP_TIMEOUT_S`).

Patch P2 — Physical `tgeompoint` fusion. P2 performs the physical fusion of the `tgeompoint` on the chain root via the MobilityDB aggregate `merge(tgeompoint[])`, so that a `valueAtTimestamp` or `atPeriod` query addressed to the root now returns the complete trajectory of all its absorbed fragments.

Idempotence guarantees. The idempotence of Stage B is guaranteed by a three-level mechanism. At the process level, a PostgreSQL advisory lock (`pg_try_advisory_lock`) enforces mutual exclusion between concurrent instances of the chain-merge. At the SQL-query level, the sentinel `fused_at` set to `NOW()` at the end of the physical-fusion phase identifies roots that have already been fused and excludes those that no longer need work. At the row level, the comparison `child.merged_at` > `root.fused_at` accurately detects chains that have gained new absorbed fragments since their last physical fusion and must therefore be re-extended, while leaving stable chains untouched. A second pass on the same window, with no new positions in between, is measured as strictly no-op (`[graduate-soft] 0`, `[edges] 0`, `[apply] 0`, `[fuse] 0`) in 0.3 s wall-clock.

Algorithm 7.4: Hourly chain-merge of `greedy_chainmerge` (Stage B, P1+P3+P2).

Input: tables `rt.stib_trip_open`, `rt.stib_trip_cold`; scanning window *window_hours* = 2.0; thresholds *gap_thr*, *dist_thr*, *max_speed* as in Table 7.4.
Output: updates of `superseded_by`, `merged_at`, `fused_at`; rows in `rt.stib_trip_merge_log`; extended `tgeompoint` on chain roots.

1. Acquire the advisory lock; abort if already held.
2. *[P3]* Move to cold every trip in `stib_trip_open` inactive for more than 300 s (five minutes), by an atomic DELETE-RETURNING-INSERT CTE.
3. Load active trips in the window: `superseded_by IS NULL`, `end_ts` $\in$ [NOW() − *window_hours*, NOW() − 15 min].
4. Group loaded trips by (ℓ, *direction*).
5. For each group, enumerate pairs (A, B) with $A.end\_ts < B.start\_ts$; compute $gap = B.start\_ts - A.end\_ts$, $dist$ = haversine($A.last\_point, B.start\_point$), $v = dist/gap$; retain the edge if $gap \in [1, gap\_thr]$, $dist \le dist\_thr$, $v \le max\_speed$.
6. Sort edges by ascending *gap*; build a Union-Find with constraints "each absorbed at most once" and "each predecessor at most once"; *[P1]* select the root by minimal (*start_ts*, *trip_id*).
7. For each retained edge, set *true_kept* ← uf.find(*kept*); `UPDATE rt.stib_trip_cold SET superseded_by = true_kept, merged_at = NOW() WHERE trip_id = absorbed AND superseded_by IS NULL`; log the fusion in `rt.stib_trip_merge_log`.
8. *[P2]* Identify roots to (re-)fuse physically: `fused_at IS NULL` or at least one absorbed with `child.merged_at` > `root.fused_at`; aggregate via `merge(tgeompoint[])` on the root after trimming overlaps and dropping shadowed pieces; update `trip`, `end_ts`, `point_count`, `last_point`, `fused_at = NOW()`.
9. Check invariants (no orphans, no self-loops) by correlated `NOT EXISTS` scoped to `merged_at >= NOW() - 10 min`.
10. Release the advisory lock.

The most prominent conceptual difference between Stage B of `greedy_chainmerge` and the anchor-aware merge of `hungarian_merge` is the deliberate absence of any GTFS-anchoring consultation. The fusion decision rests only on the triple ($gap, dist, v$) and on the (ℓ, *direction*)

Table 7.4: Production thresholds of the `greedy_chainmerge` chain-merge (Stage B).

Parameter	Value	Justification
gap_thr	1 200 s	Upper bound of a tolerable drop-out (20 min); beyond this, an operationally distinct chain is assumed.
dist_thr	1 500 m	Urban inter-station distance plus a GPS drift margin.
max_speed	25 m/s	Physical ceiling of STIB-MIVB trams and buses (∼90 km/h); above this, the fusion would be a teleport.
window_hours	2.0	Scanning window; a 15-min trailing buffer stabilises the hot edge.
Pre-graduation age	300 s	Minimal age of an open trip for the P3 pre-graduation, hard-coded as `INTERVAL '5 minutes'` in the chain-merge CTE.

key. The argument for dropping the semantic guard is the partial coverage of the static anchoring on STIB (`anch_pct` = 59.5% for `hungarian_merge`, Figure 7.2), which would otherwise make the anchor-aware guard excessively restrictive: it would block close to a third of legitimate merges to remove a much smaller fraction of false positives. The tightening of *dist_thr* (from 2 000 m down to 1 500 m) and the single upper bound on speed ($v \leq 25$ m/s with no lower floor, replacing $v \in [0.5, 15]$ m/s) partly compensate for the dropped semantic guard, while the loosening of *gap_thr* (from 600 s up to 1 200 s) reflects the typical duration of the network drop-outs observed on STIB.

`greedy_chainmerge` storage schema.

The database support for Stage B is three additional columns on `rt.stib_trip_cold`: `superseded_by TEXT` (a reference to the canonical root of the chain, forming an explicit forest constrained by `CHECK (superseded_by IS NULL OR superseded_by <> trip_id)` that forbids self-loops), `merged_at TIMESTAMPTZ` (the instant at which the `superseded_by` pointer was set), and `fused_at TIMESTAMPTZ` (idempotence sentinel local to the P2 physical fusion). A dedicated log table `rt.stib_trip_merge_log` records every retained edge with its parameters *gap_s*, *dist_m*, *v_mps* and the algorithm version. A partial index `idx_stib_trip_cold_supby` on `superseded_by WHERE superseded_by IS NOT NULL` supports the physical-fusion phase without weighing on the volume of roots. The complete DDL (foreign-key constraint and secondary indices of `rt.stib_trip_merge_log`) is reported in Appendix A.

This explicit-pointer forest was preferred over a Slowly-Changing-Dimension Type 2 pattern [64]. SCD 2 versions the attributes of one dimension row across time, whereas the chain-merge problem is to link separate trip rows that prove to be the same physical run; the `superseded_by` pointer records that linkage in a single nullable column, with no row duplication.

Synthetic comparison `hungarian_merge` → `greedy_chainmerge`. Table 7.5 summarises the eleven axes on which `greedy_chainmerge` diverges from `hungarian_merge`. The pattern is informative: every per-line dynamic component of `hungarian_merge` has been collapsed to a global fixed value, the anchor-aware guard has been removed, the post-process has been moved from inside the matcher to a separate hourly stage with its own schema support, and the canonical chain root is now selected by an explicit chronological rule rather than by an implicit first-encountered heuristic.

7.4.4 Editorial Alerts Ingestion (Traveller Info)

The bus-bunching alerts of Section 7.11 are computed in-house from the ingested vehicle positions and the geometric thresholds of `static.stib_line_thresholds`. Alongside this

Table 7.5: Eleven-axis comparison of `hungarian_merge` (Section 7.4.1) and `greedy_chainmerge` (this section).

Aspect	**`hungarian_merge` (§7.4.1)**	**`greedy_chainmerge` (this section)**
Streaming matcher	Per-batch Hungarian assignment (Algorithm 7.1)	Nearest-first greedy with curvilinear validation
Temporal gates	Per-line dynamic (cadence × peak/off-peak/night)	Global fixed (500 m, 300 s)
Curvilinear gates	Per-line dynamic (anchor mode × cadence)	Global fixed (30 m/s, 50 m)
Splitting mechanism	Per-line duration cap ($1.8 \times p_{95}(sched\_dur)$)	DENSE_SPLIT: refuse extension if $elapsed > headway \times 2.0$ (dense lines only)
Post-process	Zero-risk anchor-aware feed-dropout merge (Algorithm 7.2)	Union-Find chain-merge on $(gap, dist, v)$ with no anchoring consultation
Post-process thresholds	$gap \leq 600$ s, $dist \leq 2$ km, $v \in [0.5, 15]$ m/s, $bearing \leq 45°$	$gap \leq 1\,200$ s, $dist \leq 1\,500$ m, $v \leq 25$ m/s
Anchor-aware guard	Yes (blocks fusion if $\sigma(A) \neq \sigma(B)$)	No
Canonical chain root	Implicit (first encountered)	*[P1]* Chronological: minimal $(start\_ts, trip\_id)$
Merge scope	Cold only	*[P3]* Cold + forced pre-graduation of open trips inactive > 5 min
Physical fusion of `tgeompoint`	No (root = fragment, own positions only)	*[P2]* Aggregated `merge(tgeompoint[])`, root is a complete `SequenceSet`
Database schema	Implicit (in-place rewrite in `rt.stib_trip`)	Explicit: `superseded_by`, `merged_at`, `fused_at`, partial index, log table

algorithmic detector, STIB publishes on its Open Data portal a stream of *editorial* alerts, written by its operators: planned works, major incidents, temporary service changes, station closures. The stream is integrated by a dedicated ingestor, disjoint from the position pipeline, with its own cadence and persistence table. The two alert families, algorithmic and editorial, then coexist in the frontend under the unified TransportRT overlay (see Section 7.12).

Cadence and persistence table. The dedicated ingestor `traveller_info_ingestor.py` polls the ODP endpoint at a cadence of 600 s (10 min), configured through the constant `DEFAULT_INTERVAL = 600.0` and confirmed by the ingestion runner. This slow cadence, two orders of magnitude above the position polling cadence (20 s), reflects the slow nature of the editorial stream: editorial alerts typically live from several hours to several days, and a refresh at 10 min is enough to catch their transitions while limiting the load on the ODP.

Messages are persisted into `rt.stib_traveller_message`, whose main columns are: `message_id` (UUID, primary key, default `gen_random_uuid()`); `content_hash` (UNIQUE, computed as a SHA-1 over the multilingual text, the priority, the sorted `line_ids` and the sorted `point_ids`), used as a deduplication key to absorb republications with identical content but variable identifier on the ODP side; the multilingual fields `text_fr`, `text_nl`, `text_en`; `priority` (`SMALLINT`), the raw ODP integer typically taking values 1 through 9; `line_ids[]` and `point_ids[]`, arrays of references to the impacted lines and stops (`NOT NULL DEFAULT '{}'`); the lifecycle triple `first_seen_at`, `last_seen_at`, `resolved_at`, the last of which is set to `NOW()` on the cycle at which the alert leaves the feed; and `raw` (`JSONB NOT NULL DEFAULT '{}'`) which preserves the original ODP payload for audit and deterministic replay if a derived field needs to be re-projected.

The API endpoint `GET /api/stib/traveller_info?active=1&limit=1000` exposes the still-active messages (`resolved_at IS NULL`) to the frontend; it is queried in particular by the P1/P2 alert bell (Section 7.12) and by the `prio` pills of the subfilter (Section 7.12).

Mapping `priority` → `niveau`. The frontend reads a homogeneous `niveau` $\in \{1, 2, 3\}$ field that aggregates the two operators behind the same semantics. On the STIB side, this field is *not* a one-to-one projection of the ODP `priority` field, which takes nine distinct values in typical operation. The Flask service `stib_service.py` applies a three-level discretisation by a tripartite `CASE WHEN` expression: values 1, 2 and 3 map to `niveau` 1 (high); 4, 5, 6 map to `niveau` 2 (medium); 7, 8, 9 map to `niveau` 3 (low). The projection collapses an initially finer scale onto three classes, aligning the semantics with the one used on the De Lijn side (Section 7.6.1) and simplifying the frontend filter logic (`prio` pills, P1/P2 alert bell, TTS trigger).

7.5 Empirical Validation of the Trip Reconstruction Matcher

The matcher of Section 7.4.1 is one of several plausible designs for stateless-feed trip reconstruction. The most natural external baseline is the open-source vehicle-identification algorithm maintained as part of the MobilityTwin.Brussels Components project [65], which also targets the STIB-MIVB feed and is the direct reference design for any work that proposes to replace it.

This section benchmarks the seven matchers of the design lineage, from the `greedy_haversine` baseline to the deployed `greedy_chainmerge`, against that vendored baseline, on a single seven-day capture of the live STIB feed under identical input and identical hardware, following the protocol of Section 5. The benchmark exercises the full ingestion-to-reconstruction pipeline (Sections 7.4.1, 7.7 and 7.8); only the matcher component differs between runs, so the differences in quality metrics and runtime reported below are attributable to the matcher itself.

One caveat is stated up front and repeated on every figure: the measurements are point estimates over a single seven-day window, with no run-to-run variance; a bootstrap over hourly slices remains a future-work item (Section 11).

The schedule-anchoring step. Throughout this section the term *anchoring* refers to a fixed post-process applied identically to every matcher under test: each reconstructed trip τ is associated with at most one scheduled trip identifier $\sigma(\tau) \in \mathcal{S} \cup \{\perp\}$ from the GTFS static feed overlapping the capture window. The association is computed by `anchor_trips_to_schedule` (`src/comparison/mobilitytwin/post_anchor.py`) on three agreement criteria: the reconstructed trip and the candidate scheduled trip share `(route_id, direction_id)`; their time ranges have intersection-over-union above the line-specific cadence threshold (≥ 0.5 on bus and tram, ≥ 0.6 on metro); and the spatial coverage score [52] of the reconstructed trip along the scheduled trip's stop sequence exceeds a confidence floor of 0.7. When several scheduled trips qualify, the one minimising the absolute start-time gap is retained; otherwise $\sigma(\tau) = \perp$. The anchor-aware guard of Algorithm 7.2 reads σ exactly as defined here; the anch_pct metric counts the trips with $\sigma(\tau) \neq \perp$.

7.5.1 Setup

Capture window. The reference dataset is a continuous capture of the live STIB feed over the seven days from 2026-05-04 to 2026-05-10. The window spans the full weekly cycle, so it includes ordinary weekday operation alongside the weekend, and it carries the natural mix of peak and off-peak headways that a single short capture cannot. The GTFS static feed overlapping the window contains 78 921 scheduled courses; this is the schedule-derived reference against which the reconstructed trip counts are measured. Coverage spans the STIB metro, tram and bus network.

Baseline. The MobilityTwin baseline is the `IdentifyVehicleAlgorithm` class taken verbatim from `components/stib/harvesters/identify_vehicle/algorithm.py` of the AITwin/Components repository [65], vendored on 2026-04-29 from the `main` branch under its CC BY-NC-SA 4.0 license. The algorithm is architecturally distinct from the lineage: it groups observations by (*lineId*, *direction*), runs a greedy nearest-vehicle match per group with batch overlap, then runs a z-score-based split detector (threshold $|z| > 4$ on inter-observation speeds) followed by a heuristic merge step. It does not validate progress along the line shape, does not consult the GTFS schedule for terminus-aware splitting, and does not anchor its output to scheduled trip identifiers; the schedule-anchoring step is therefore applied to its output post-hoc, with the same anchoring code, so that it differs from the lineage matchers only in the matcher itself.

The matcher lineage. Seven matchers of the design lineage are replayed on the same capture: `greedy_haversine`, the Haversine-only greedy assignment of the v1 loader; `curvilinear`, which adds the along-shape progression gate; the four Hungarian-assignment refinements `hungarian`, `hungarian_snap`, `hungarian_terminus` and `hungarian_merge`; and `greedy_chainmerge`, the two-stage matcher deployed in production (Section 7.4.3). The vendored MobilityTwin algorithm is the eighth competitor, and the GTFS schedule is reported as a synthetic reference of perfect faithfulness. The per-step contribution of each refinement is examined in Appendix E.

Hardware and methodology. The benchmark runs on the benchmark VM (Section 5.1). Every matcher shares the same input loader, the same anchoring step and the same timing harness. Reported runtimes are wall-clock figures for the seven-day run.

7.5.2 Quality Metrics

The metric definitions below are reproduced from `src/comparison/mobilitytwin/quality_metrics.py`. Let $\mathcal{T}$ be the set of reconstructed trips, partitioned by line ℓ as $\mathcal{T}_\ell$, and let $\mathcal{S}$ be the GTFS scheduled trips on the same window with line counts $n_{\text{gtfs}}(\ell)$. For each $\tau \in \mathcal{T}$, let $obs(\tau)$ be its observations, ordered in time.

The count_score is the line-wise median of the ratio between reconstructed and scheduled trip counts:

$$\text{count\_score} \;=\; \text{median}_{\ell : n_{\text{gtfs}}(\ell) > 0} \frac{|\mathcal{T}_\ell|}{n_{\text{gtfs}}(\ell)}, \tag{7.1}$$

so a perfect matcher (one reconstructed trip per scheduled trip on every line) yields 1, and a matcher that emits k times too many trips yields k.

frag_pct is the share of fragmented trips:

$$\text{frag\_pct} \;=\; \frac{100}{|\mathcal{T}|} \cdot \big|\{\tau \in \mathcal{T} : |obs(\tau)| \le 3\}\big|. \tag{7.2}$$

hi_speed_pct (the physical-plausibility canary) is the share of consecutive within-trip observation pairs whose implicit speed exceeds 25 m/s (90 km/h):

$$\text{hi\_speed\_pct} \;=\; \frac{100}{|\Pi|} \cdot \Big|\Big\{(o_i, o_{i+1}) \in \Pi : \tfrac{|\Delta s|}{\Delta t} > 25\,\text{m/s}\Big\}\Big|, \tag{7.3}$$

with Π the set of within-trip ordered pairs of observations. A non-zero value flags physically implausible reconstructions.

median_cov_pct is the line-wise median of the per-trip stop-coverage rate among anchored trips:

$$cov(\tau) \;=\; \frac{|\{s \in stops(\sigma(\tau)) : \min_{o \in obs(\tau)} \text{haversine}(o, s) \le 80\,\text{m}\}|}{|stops(\sigma(\tau))|}, \tag{7.4}$$

$\mathsf{median\_cov\_pct} = 100 \cdot \mathrm{median}_{\tau:\sigma(\tau)\neq\perp}\, cov(\tau)$.

Finally, clean_pct is the share of trips that begin *and* end inside the terminus point-id set T_ℓ of their line (Appendix D):

$$\mathsf{clean\_pct} \;=\; \frac{100}{|\mathcal{T}|} \cdot \left|\{\tau \in \mathcal{T} : \mathit{first\_pid}(\tau) \in T_{\ell(\tau)} \wedge \mathit{last\_pid}(\tau) \in T_{\ell(\tau)}\}\right|. \tag{7.5}$$

A further quantity, the schedule-anchoring rate anch_pct (the share of reconstructed trips with $\sigma(\tau) \neq \perp$), is produced by the benchmark but excluded from the quality verdict. Figure 7.2 shows why: anch_pct rises with the fragmentation rate rather than with reconstruction quality. The most fragmented matchers, `greedy_haversine` and MobilityTwin, score around 90% because their many short fragments accidentally satisfy the temporal-overlap criterion of the anchoring step, while `greedy_chainmerge`, which emits long and faithful trips, sits near 40%. A high anch_pct is therefore a fragmentation symptom, not a quality advantage, and the metric is read only as a diagnostic.

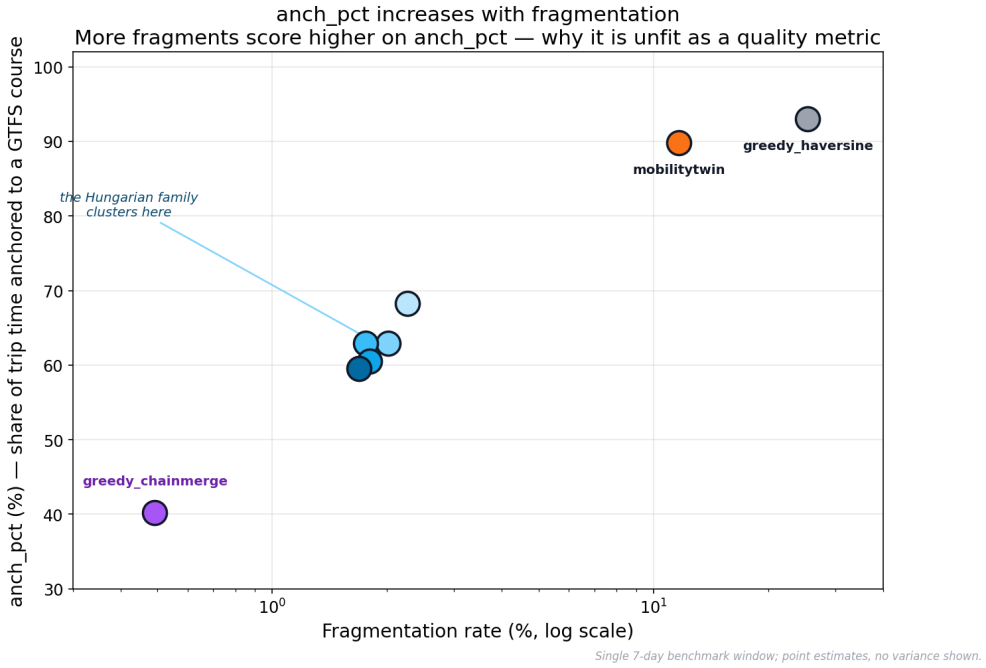


Figure 7.2: Schedule-anchoring rate anch_pct against the trip fragmentation rate on the seven-day capture. The rate rises with fragmentation, not with quality: the most fragmented matchers score highest because their short fragments accidentally satisfy the temporal-overlap criterion of the anchoring step. anch_pct is therefore excluded from the quality verdict. Single seven-day benchmark window; point estimates.

7.5.3 Results

Figure 7.3 traces three metrics along the design lineage on the seven-day capture. Median trip coverage rises in two large steps, at the `curvilinear` gate and again at `greedy_chainmerge`, with the Hungarian-family matchers forming a plateau between the two; the fragmentation rate falls across the lineage from 25.4% to 0.49%; and the impossible-speed rate drops sharply at the `curvilinear` step before rising again on the last two matchers, a pattern returned to in Section 7.5.5.

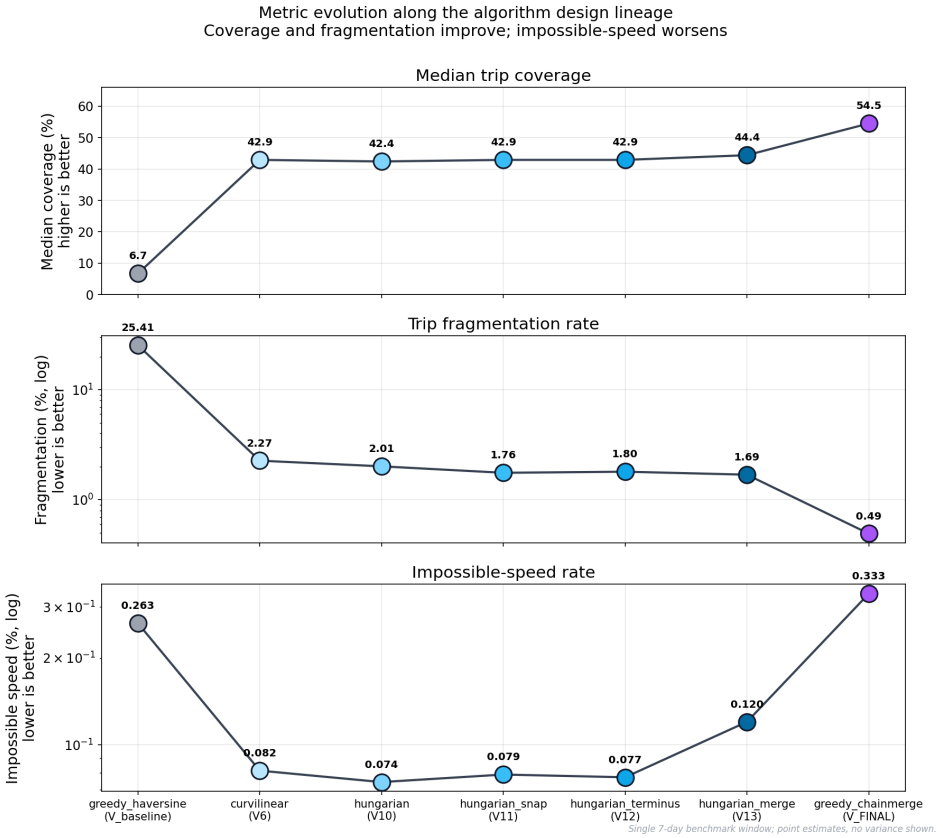


Figure 7.3: Three quality metrics along the matcher design lineage, from `greedy_haversine` to the deployed `greedy_chainmerge`, measured on the seven-day STIB capture. Median trip coverage (top) and the trip fragmentation rate (middle) improve along the lineage; the impossible-speed rate (bottom) falls at the `curvilinear` gate and rises again on the last two matchers. Single seven-day benchmark window; point estimates.

Table 7.6 and Figure 7.4 report the four quality metrics across the eight matchers. `greedy_chainmerge` leads three of them: a median trip coverage of 54.5% against 44.4% for `hungarian_merge` and 21.4% for MobilityTwin; a fragmentation rate of 0.49%, the only matcher below one percent, against 1.69% and 11.64% respectively; and a clean-trip share of 25.4% against 22.8% and 13.9%. On `count_score` it is the closest to the ideal of 1 (1.34, against 2.32 for `hungarian_merge` and 5.77 for MobilityTwin), the line-wise signature of a matcher that emits close to the right number of trips per line. The same metrics are reproduced per version in Appendix E, in a table that adds one diagnostic column, the schedule-anchoring rate `anch_pct`.

The raw trip count tells the same story at the network scale. `greedy_chainmerge` reconstructs 97 807 trips against the GTFS reference of 78 921 scheduled courses, a factor of 1.2; `hungarian_merge` emits 156 520 (2.0×), MobilityTwin 342 709 (4.3×), and the `greedy_haversine` baseline 495 419 (6.3×), each over-segmenting the feed into far more trips than the schedule contains. Figure 7.5 plots the eight counts against the reference.

Figure 7.6 places each matcher in the coverage-versus-fragmentation plane, with the GTFS reference at the ideal corner (top-left: full coverage, no fragmentation). `greedy_chainmerge` is the closest of the eight to that corner. The five Hungarian-family matchers cluster tightly on a coverage plateau between 42 and 44%, at a fragmentation between 1.7 and 2.3%: the refinements from `hungarian` to `hungarian_merge` reduce fragmentation without lifting coverage off the plateau. `greedy_chainmerge` is the only matcher that leaves the plateau, and it does so

Table 7.6: Quality metrics across the eight matchers on the seven-day STIB capture (GTFS reference: 78 921 scheduled courses). count_score is ideal at 1; arrows give the direction of improvement. Values computed from the seven-day benchmark `compare.csv`.

Algorithm	n_trips	count_score→ 1	frag_pct↓	hi_speed_pct↓	median_cov_pct↑	clean_pct↑
`greedy_haversine`	495 419	8.01	25.41	0.263	6.7	20.8
`curvilinear`	165 571	2.30	2.27	0.082	42.9	22.0
`hungarian`	164 192	2.24	2.01	**0.074**	42.4	22.1
`hungarian_snap`	167 792	2.61	1.76	0.079	42.9	19.5
`hungarian_terminus`	161 415	2.46	1.80	0.077	42.9	21.0
`hungarian_merge`	156 520	2.32	1.69	0.120	44.4	22.8
`greedy_chainmerge`	**97 807**	**1.34**	**0.49**	0.333	**54.5**	**25.4**
MobilityTwin	342 709	5.77	11.64	2.149	21.4	13.9
GTFS reference	78 921	1.00	0.14	0.000	100.0	100.0

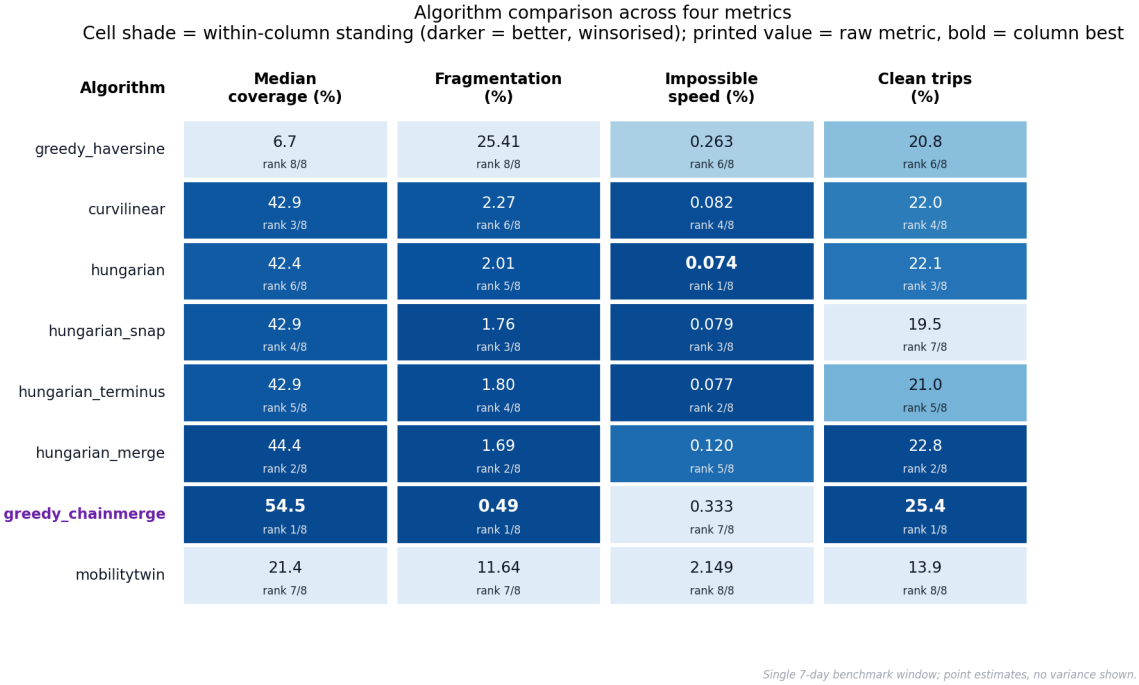


Figure 7.4: Four quality metrics across the eight matchers on the seven-day STIB capture: median trip coverage, trip fragmentation rate, impossible-speed rate and clean-trip share. Cell shading is the within-column standing; the printed value is the raw metric. `greedy_chainmerge` leads on coverage, fragmentation and clean-trip share. Single seven-day benchmark window; point estimates.

by a clear margin on both axes. MobilityTwin and `greedy_haversine` sit far from the corner, an order of magnitude worse on fragmentation. The MobilityTwin algorithm is benchmarked here only as deployed. Whether injecting the curvilinear topological gate into it would narrow the gap on fragmentation was not tested, and remains for future work.

7.5.4 Quality against Cost

The coverage gain is not free. Figure 7.7 reports the end-to-end runtime of each matcher over the seven-day capture. `greedy_chainmerge` completes in 6.5 min, split into 1.7 min of streaming

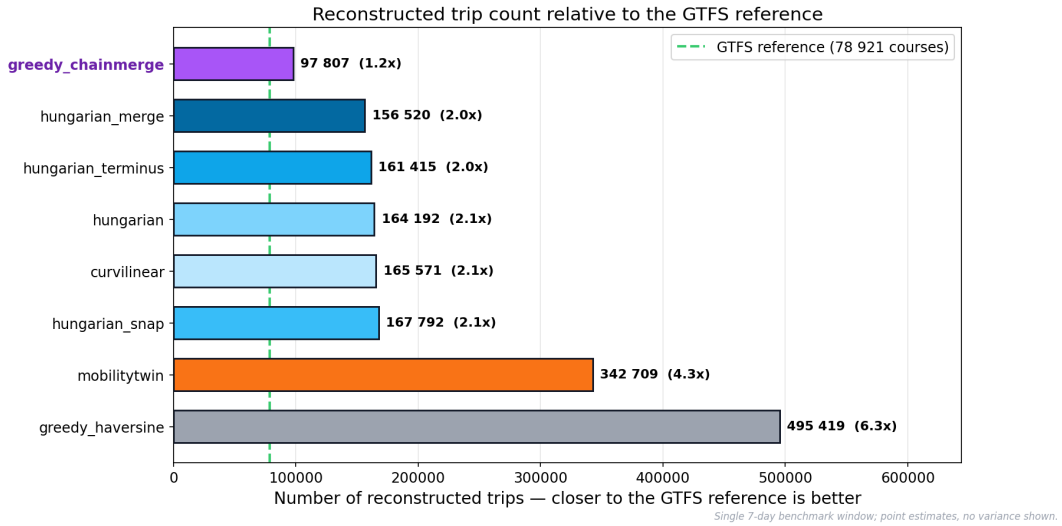


Figure 7.5: Reconstructed trip count per matcher on the seven-day capture, against the GTFS reference of 78 921 scheduled courses (dashed line). `greedy_chainmerge` is the closest to the reference at a factor of 1.2. Single seven-day benchmark window; point estimates.

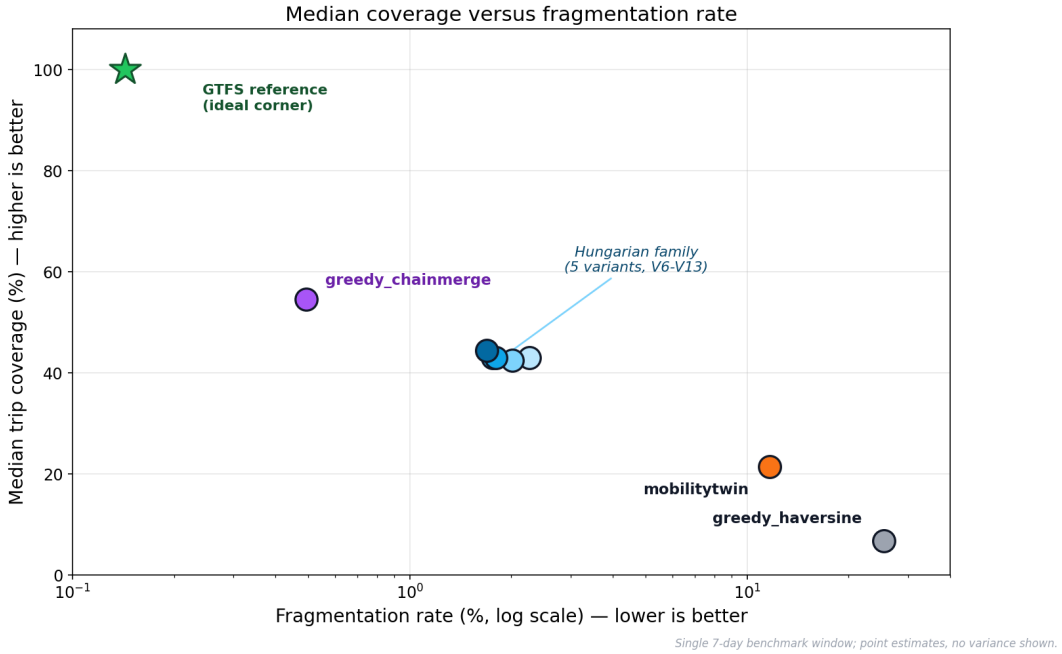


Figure 7.6: The eight matchers in the coverage-versus-fragmentation plane (fragmentation on a log scale). The GTFS reference marks the ideal corner. `greedy_chainmerge` is the closest of the eight; the Hungarian-family matchers cluster on a coverage plateau. Single seven-day benchmark window; point estimates.

Stage A and 4.8 min of hourly chain-merge, against 3.8 min for `hungarian_merge` and between 1.4 and 2.4 min for the five earlier matchers. It is the slowest of the lineage, yet it stays more than twenty times faster than the 144.9 min of MobilityTwin, whose per-group greedy assignment with batch overlap dominates its cost.

Figure 7.8 weighs quality against that cost. On the coverage-versus-runtime plane `greedy_chainmerge` is the sole non-dominated point at high coverage: no matcher reaches

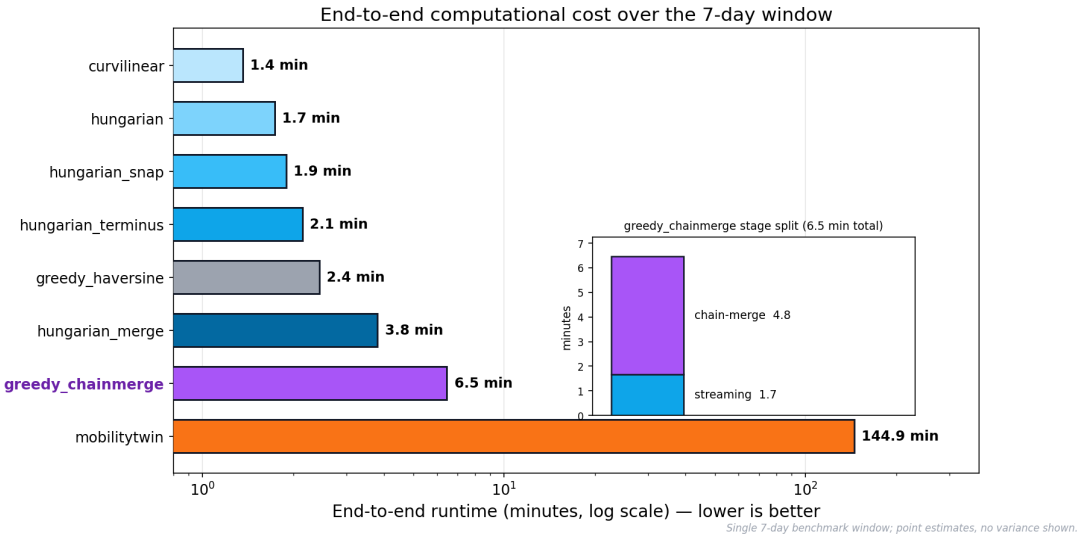


Figure 7.7: End-to-end runtime per matcher over the seven-day capture (log scale). `greedy_chainmerge` is the slowest of the lineage at 6.5 min (inset: 1.7 min streaming, 4.8 min chain-merge), yet it stays more than twenty times faster than MobilityTwin. Single seven-day benchmark window; point estimates.

its coverage at a lower runtime. Over a seven-day batch the 6.5-min figure is operationally negligible, so the chain-merge cost is paid willingly for the coverage it returns.

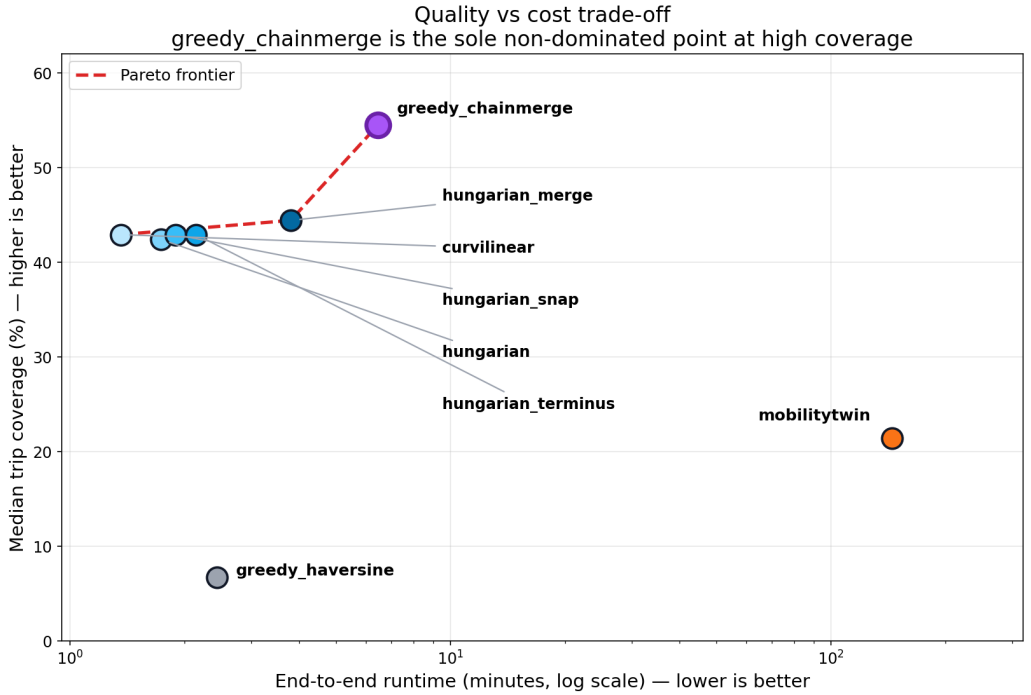


Figure 7.8: Median trip coverage against end-to-end runtime over the seven-day window (runtime on a log scale). `greedy_chainmerge` is the sole non-dominated point at high coverage: no matcher reaches its coverage at a lower runtime. Single seven-day benchmark window; point estimates.

7.5.5 The Chain-Merge Mechanism

Figure 7.9 follows where `greedy_chainmerge`'s coverage gain comes from. The Stage A streaming matcher emits 255 599 fragments over the seven days. The hourly chain-merge then applies 134 813 fusions, collapsing the forest to 120 786 chains, and post-filtering removes the residual short fragments to leave 97 807 reconstructed trips. The fused gaps are short and physically plausible: the mean bridged gap is 354 s over 624 m, an implied chord speed of 4.6 m/s (17 km/h), well inside the 25 m/s plausibility ceiling. The mechanism is the offline, lineage-wide counterpart of the production fusion forest observed in Section 7.5.8.

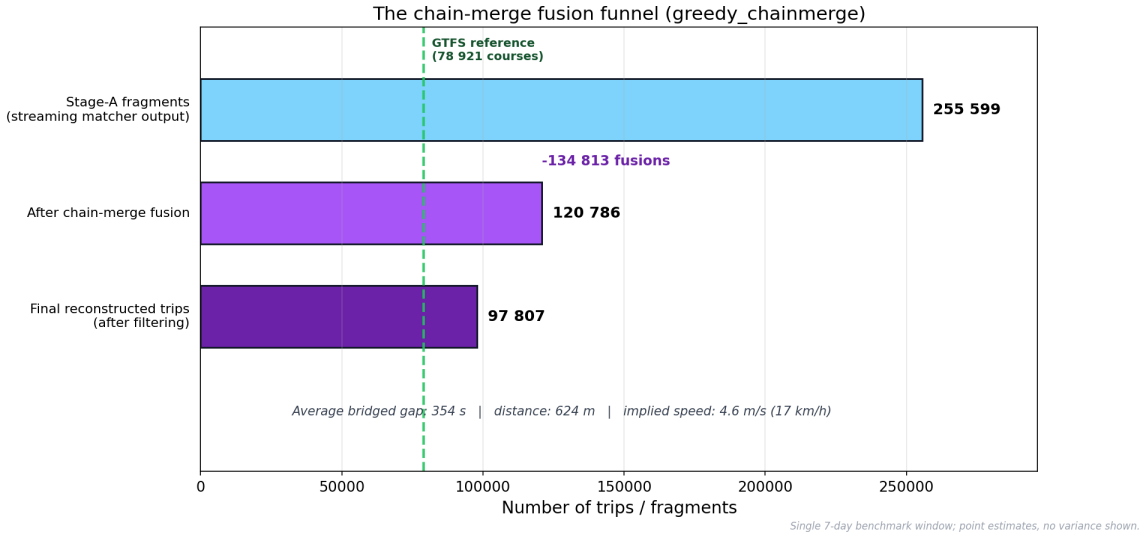


Figure 7.9: The chain-merge fusion funnel of `greedy_chainmerge` over the seven-day window. The Stage-A streaming matcher emits 255 599 fragments; the hourly chain-merge applies 134 813 fusions; post-filtering leaves 97 807 reconstructed trips, against the GTFS reference of 78 921 courses. Single seven-day benchmark window; point estimates.

The impossible-speed trade-off. One metric does not follow `greedy_chainmerge`'s lead. Its impossible-speed rate is 0.333%, above the 0.07 to 0.12% band of the Hungarian family, above the 0.1% reference line used as a plausibility canary in Appendix E, and second only to the 2.149% of MobilityTwin. The rise is plausibly attributable to the Stage A streaming matcher, which reverts to a nearest-first greedy assignment (Section 7.4.3) that is less conservative on dense batches than the Hungarian solver retained in `hungarian_merge`. The rate stays an order of magnitude below MobilityTwin, but it is the one axis on which the production specialisation trades physical plausibility for coverage rather than improving both together, and it is reported here rather than smoothed over. A targeted audit of the high-speed transitions, and a tightening of the streaming-stage gates if they prove avoidable, is added to the follow-up agenda of Section 11.

7.5.6 Where the Gap Comes From

The order-of-magnitude separation between the lineage and MobilityTwin warrants a causal explanation rather than a numerical one.

One alternative explanation can be set aside first. Figure 7.10 plots, for each matcher, the volume of GPS observations ingested against the number of reconstructed trips. Every matcher consumes a comparable observation volume, between 6.3 and 8.2 million over the seven days; the wide spread in trip count, from 97 807 to 495 419, is therefore a property of the matcher, not of

uneven input. With the input held constant, three design differences account for the gap.

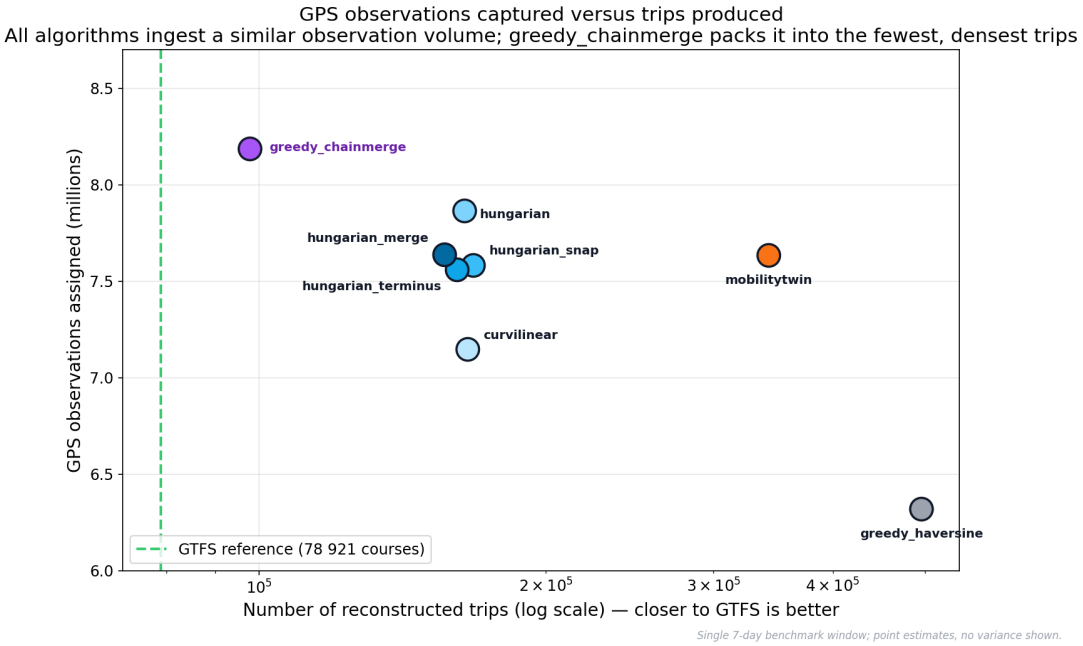


Figure 7.10: GPS observations ingested against reconstructed trip count, per matcher, on the seven-day capture. Every matcher consumes a comparable observation volume; the spread in trip count is therefore produced by the matcher, not by uneven input. Single seven-day benchmark window; point estimates.

Curvilinear versus statistical splits. MobilityTwin detects "strange trips" via a z-score on inter-observation speeds, which works on highway-scale GPS feeds where speed is approximately stationary. On a dense urban tram or bus feed, speeds vary by an order of magnitude over a single trip (stop dwell versus between-stop sprint), and the z-score discrimination collapses: legitimate stops are flagged as splits, and true outliers, such as a platform crossover at a hub, escape the threshold. The curvilinear gate, introduced at the `curvilinear` step and carried by every later matcher, replaces this with a direct progression-along-the-shape test, which generalises across the three modes without per-line tuning. Its effect is the largest single step in the lineage: fragmentation falls from 25.41% to 2.27%.

Schedule-aware versus schedule-blind closure. A bus standing for five minutes at a terminus before starting its return trip is a single open trip ending and a fresh trip beginning, not one mega-trip and not three fragments. MobilityTwin has no representation of "terminus" and therefore cuts a typical layover into one to three sub-trips depending on the dwell time and feed cadence. The terminus-aware cap, introduced at `hungarian_terminus`, consults the precomputed line-stops set and lets a layover survive as a single boundary between two clean trips.

Anchor-feedback and chain-merge. The lineage closes feed dropouts with a post-process that the baseline lacks. `hungarian_merge` consults the schedule-anchoring output before fusing two halves of an apparent dropout: if the halves anchor to two distinct scheduled trips, the merge is suppressed; on the seven-day capture this guard skips 3 725 such merges. `greedy_chainmerge` replaces the anchor-aware guard with the geometric chain-merge of Section 7.5.5, a change motivated by the production observation that schedule anchoring covers only about 60% of live trips. MobilityTwin has no equivalent loop: its post-hoc heuristic merge cannot tell "same vehicle resumes after dropout" from "next scheduled trip starts where the previous one ended".

Threats to validity. Two threats remain. The benchmark covers a single seven-day window: the weekly cycle is represented once, with no run-to-run variance, and a bootstrap over hourly slices, which would attach confidence intervals to the per-metric deltas, was not run. And the lineage, in particular the chain-merge thresholds, was tuned on STIB; recalibration would be required on a feed with a different cadence or terminus topology. The De Lijn feed carries a stable `vehicle_id` and does not need the matcher at all; cross-operator validation on another stateless feed is part of the future-work agenda (Section 11).

Verdict. On a seven-day window that includes ordinary weekday traffic, `greedy_chainmerge` reconstructs trips at a higher coverage, a lower fragmentation, a higher clean-trip share and a count closer to the GTFS reference than any other matcher in the lineage, and far ahead of the vendored MobilityTwin baseline on every quality axis. It pays for this with a higher impossible-speed rate and a runtime that, although the largest of the lineage, stays trivial against the seven-day window. The result supports the engineering choice of Section 7.4.3 to deploy `greedy_chainmerge` in continuous production.

7.5.7 Camera Ground-Truth Benchmark

Every metric of Section 7.5 is computed against the GTFS schedule or from the matchers' own output; none of them observes the street. An external ground truth was collected on 2026-05-05 during an invited session in the operations room of Bruxelles Mobilité. Six Genetec cameras — Etterbeek Gare, Rodin, Trône, Tour & Taxi, Porte de Namur and Buyl — were watched over a 12:30–19:30 window, and every transit vehicle that passed was logged by hand as a `(camera, time, line, direction)` record.

The cameras were chosen on dense corridors (lines 8, 25, 71, 95, 46, 54 and 81) for observation volume, and among the feeds the operators were not using at the time, so the collection did not interfere with their work. The session produced 135 manual records; 125 remain after dropping ten De Lijn R-line sightings, since the position archive for that day holds STIB only. Figure 7.11 shows one frame of the material these records were read from.

An observation counts as matched when at least one reconstructed trip passes within an adaptive 50–80 m buffer of the camera, inside the window $[T_{\text{obs}} - 120\,\text{s},\ T_{\text{obs}} + 240\,\text{s}]$, and in the observed direction; the matched share is the camera-match rate `cam%`. The uniqueness key is the pair `(trip, camera)` rather than `trip` alone. A long, correct trip physically crosses several of the six cameras, so it is allowed to match one observation at each — but at most one per camera, so that two sightings at the same camera, which are two distinct passages, cannot both be claimed by a single trip.

The direction test relies on a terminus-code alias built outside the pipeline, since the live feed emits direction codes that around twenty-five lines do not share with the static terminus table. Trips are reconstructed over the full day (00:00–24:00, 1 347 422 raw STIB positions), not over the observation window alone, so the score reflects the matcher as it runs in production. The zero-progress and off-route quality filters of the lineage, both at a 150 m threshold, are applied uniformly to every matcher before scoring, so `cam%` reflects reconstruction quality and not a raw fragment count. Table 7.7 gives the result.

The deployed matcher `greedy_chainmerge` reaches `cam%` = 92.0% (115/125), tied with `hungarian_snap` and `hungarian_terminus` and one match below the vendored MobilityTwin baseline (92.8%). The eight matchers spread from 65.6% for the `greedy_haversine` baseline to 92.0% along the design lineage of Section 7.4.3: the baseline is last by a wide margin and the production matcher sits in the leading group, the same broad ordering the seven-day benchmark

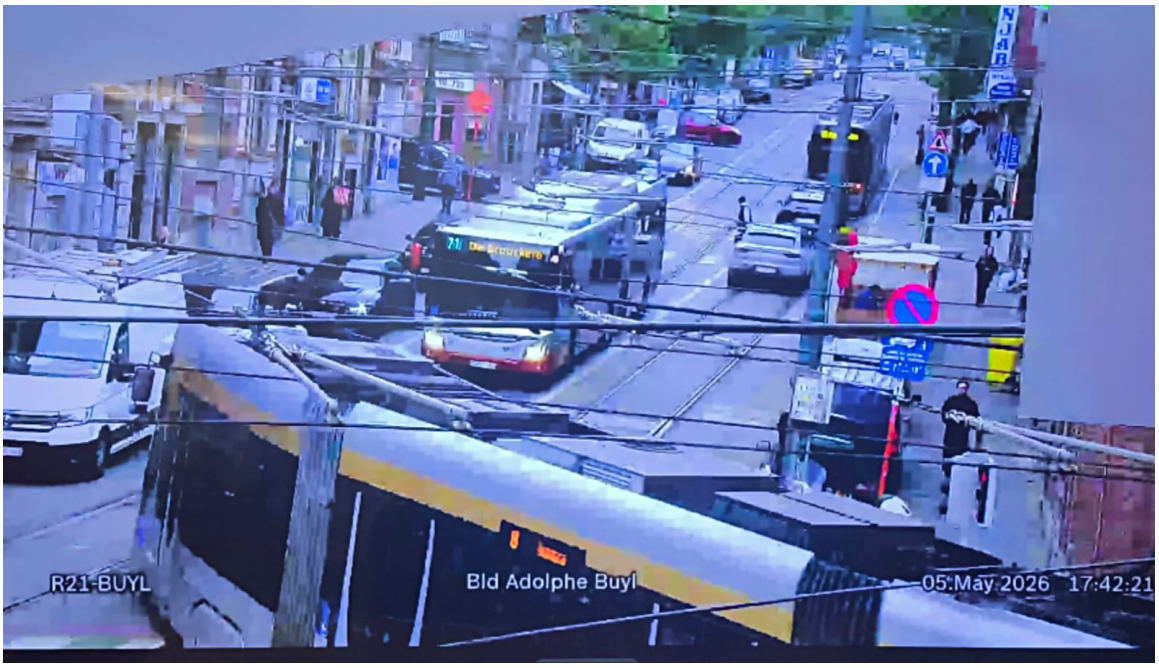


Figure 7.11: A frame from the Genetec camera R21-BUYL (Boulevard Adolphe Buyl), captured in the operations room of Bruxelles Mobilité on 2026-05-05 at 17:42 local time during the ground-truth collection session. Visible are a line-71 bus bound for De Brouckère and two line-8 trams. Each ground-truth record is one `(camera, time, line, direction)` observation read from this kind of feed; the collection logged only line, time and direction, no personal data, and individuals and vehicles are not identifiable at this resolution.

Table 7.7: Camera-match rate cam% of the eight matchers and the GTFS reference against 125 manual STIB observations collected on 2026-05-05 from six Genetec cameras. An observation is matched when a reconstructed trip crosses the camera buffer in the right time window and direction, under the `(trip, camera)` uniqueness key. Rows ordered by cam%. Single-session manual ground truth.

Algorithm	Lineage stage	cam%	Matched
MobilityTwin	vendored external baseline	92.8	116/125
`hungarian_snap`	Hungarian + headway window	92.0	115/125
`hungarian_terminus`	Hungarian + terminus snap	92.0	115/125
`greedy_chainmerge`	production matcher	**92.0**	**115/125**
`hungarian`	plain Hungarian assignment	90.4	113/125
`curvilinear`	curvilinear progression gate	87.2	109/125
`hungarian_merge`	Hungarian + feed-dropout merge	87.2	109/125
`greedy_haversine`	greedy Haversine baseline	65.6	82/125
GTFS reference	planned schedule	27.2	34/125

produced against the GTFS reference.

MobilityTwin's one-match lead does not survive the other axes. cam% is a recall metric — it asks only whether some reconstructed trip reached the camera, not whether the reconstruction is clean — and MobilityTwin earns its 92.8% by over-segmentation: it emits 3.5× more trips than `greedy_chainmerge`, at roughly 24× the fragmentation and under half the median stop coverage (Table 7.6), so one fragment or another crosses almost every camera buffer. The camera benchmark therefore corroborates the seven-day verdict instead of competing with it: an external

observer confirms that `greedy_chainmerge` matches the best matchers on recall, while the quality axes keep it ahead of MobilityTwin where production deployment needs it.

Threats to validity. The ground truth is small and single-session: 125 observations from one afternoon, because manual visual collection needs recurring access to the operations room. The nine or so observations that the best matchers never reach are not algorithm failures. They are collisions at one camera inside a few minutes — lines 25, 54 and 71 at Buyl and Porte de Namur — where three sightings compete for two or fewer distinct candidate trips. The `(trip, camera)` key forbids one trip from matching two observations at the same camera, which caps the attainable rate near 93%; relaxing it would reintroduce the over-fusion bias the key was added to catch. A multi-day collection across more cameras is left to the future-work agenda (Section 11).

7.5.8 Preliminary Production Observation of `greedy_chainmerge`

The head-to-head benchmark of Section 7.5 admitted `hungarian_merge` to production deployment. The operational specialisation `greedy_chainmerge` introduced in Section 7.4.3 replaces, in continuous service, the anchor-aware merge of `hungarian_merge` with a two-stage streaming matcher and an hourly chain-merge. The present subsection reports the empirical observation of `greedy_chainmerge` on the production window of 2026-05-16, which covers the nominal service restart at 00:30 CEST and the retroactive backfill of the three patches P1, P3, P2 at 16:00 CEST. The figures reported below are measured on the 2026-05-16 offline backfill run; the head-to-head comparison of `greedy_chainmerge` against `hungarian_merge` on a seven-day window is reported in Section 7.5.3, and continuous observation under the patched production service is listed as a future-work item in Section 11.

Activation and first cycle. The `rtdatahub-etl-pipeline` service was restarted on 2026-05-16 at 00:30 CEST with `STIB_DENSE_SPLIT_FACTOR` = 2.0 and `STIB_CHAIN_MERGE_INTERVAL_S` = 3600. The internal sentinel `_last_chain_merge_ts` initialised at 0.0 at startup triggered the first chain-merge cycle immediately. That cycle loaded 5 502 active trips in the [NOW() − 2 h, NOW() − 15 min] window of `rt.stib_trip_cold` and identified 2 738 fusions, applied as 2 738 `UPDATE`s on `superseded_by`.

Consolidated measurements after the P1+P3+P2 backfill. After the three patches were applied and the retroactive backfill on the pre-existing chains completed, followed by a full production cycle, the consolidated measurements are those of Table 7.8.

The raw fusion rate of ∼37% of the cold volume reported in the second row of Table 7.8 absorbs the effect of the retroactive backfill on the pre-patch history and is therefore not representative of the per-cycle stationary rate. The latter, measured since the patches were applied, settles in the 2–5% range of the trips active in the two-hour window, a figure consistent with the pre-`greedy_chainmerge` forensic classification reported in Appendix E (∼8.4% of zero-risk candidates in the feed drop-out class). The distinction between cumulative and stationary rate matters here to avoid reading the cumulative figure as a signal of aggressive over-fusion.

Individual effects of the three patches. The effect of P3 (forced pre-graduation) is directly quantifiable on the scope of the main scan. At a typical production instant, `stib_trip_open` contains about 2 200 trips of which roughly 1 300, or 60%, have been inactive for more than five minutes and are therefore ineligible for any further extension by the streaming matcher. Moving this subpopulation to cold ahead of the scan widens the chain-merge candidate pool by 5% to 10% per cycle. Without P3, these trips would wait for the nominal 30-min graduation, i.e. up to three hourly cycles of latency before becoming eligible for the merge.

The effect of P2 (physical fusion of the `tgeompoint`) is measured on the MobilityDB repre-

Table 7.8: Consolidated measurements of `greedy_chainmerge` on the 2026-05-16 offline backfill run.

Metric	Value	Interpretation
Cold trips over 24 h	58 353	Nominal volume.
Absorbed trips (`superseded_by IS NOT NULL`)	21 532	∼37% of the cold volume; cumulative figure including retroactive backfill.
Canonical roots (`fused_at IS NOT NULL`)	7 913	Chains physically fused.
Mean cascade (absorbed per chain)	2.7	Long-tail distribution.
Full production cycle cost	4–8 s wall	Scan 2 708 trips + 857 fusions + 220 re-fused roots.
One-shot P2 backfill cost (7 752 chains)	8.9 s	TOAST ∼18 MB; dead tuples 0.04% of the heap.
No-op cycle cost (idempotence)	0.3 s	Sentinel `fused_at` + comparison `merged_at > fused_at`.
P3 measured gain (pre-graduation)	+1 028 trips/cycle	Open trips inactive > 5 min moved to cold before the scan.
Historical debt (non-chronological kept)	1 476 / 21 532 = 6.8%	Pre-P1 fusions; root chosen by Union-Find rank.

sentation of the roots. Before P2, the root of a chain contained only its own positions, and a query `valueAtTimestamp(canonical.trip, T)` for an instant T falling in the window of an absorbed fragment returned NULL even though the position was stored in the database under the absorbed identifier. After P2, the root holds a multi-sequence `tgeompointSeqSet` that preserves the pieces and their gaps: 100% of the 7 913 fused canonical roots are of subtype `SequenceSet`, and spatio-temporal queries now return the complete trajectory of the chain.

The effect of P1 (chronological Union-Find) is measured indirectly through the *historical debt*. On the 21 532 cumulated fusions, 1 476 have a canonical with *start_ts* strictly greater than the minimum of the *start_ts* values of their absorbed fragments. These 6.8% of fusions correspond to the pre-P1 heritage, where the Union-Find rank-based root selection could retain a link that was not first in time. Every fusion applied after P1 is chronologically deterministic by construction. A one-shot SQL script, reported in the companion repository, re-canonicalises the historical debt by pointing each absorbed fragment to the earliest trip in time of its component and invalidating `fused_at` to force a consistent physical re-fusion at the following cycle.

Observed fusion distribution. The 21 888 cumulated fusions of the observed window, characterised by their gap *gap*, their distance *dist* and the implied speed v of each bridged interval, stay clear of the production thresholds: no fusion crosses $gap = 1\,200$ s, $dist = 1\,500$ m, or the iso-speed $v = 25$ m/s, which confirms the consistency of the implementation with the specification. The gap distribution shows sharp concentrations at 60, 120 and 180 s, the signature of the temporal quantisation imposed by the 20 s poll cadence of the STIB feed.

The cascade depth in the `superseded_by` forest decomposes into 19 825 fragments absorbed at depth 1 (90.5%), 2 052 at depth 2 (9.4%), and 11 cases at depth 3 (0.05%). The structural property expected by construction of the `superseded_by IS NULL` filter on the main scan (depth ≥ 3 should be impossible) is therefore respected to the order of 5×10^{-4}. The 11 residual cases suggest that a post-merge Union-Find cascade has bypassed the filter on the occasion of a cross-cycle dynamic, when a trip that becomes a root at one cycle finds itself fused into another at a subsequent cycle. This residue is the subject of an explicit item in Section 11.

Defects identified and corrected during patch validation. For methodological traceability, three defects were identified and corrected during the validation session of patches P1, P3, and P2 on

2026-05-16. They share a notable characteristic: each remained invisible at the initial dry-run and at the first isolated `-fuse-only` pass, and manifested only at the second cycle or at the transition to production scale. This observation argues for a three-stage validation protocol (dry-run, then isolated fuse-only on a small window, then full cycle on a production window) before any `systemd` deployment.

The first defect surfaced as a `psycopg.errors.InternalError_: Span cannot be empty` at the P2 physical-fusion phase of the first backfill. The cause is a collision on *start_ts* between two pieces of the same chain (typically two observations to the second on a dense line): the LEAD window produced `next_start = start_ts` for the current piece, and MobilityDB rejected the resulting empty span $[T, T)$. The fix introduces an explicit guard in the trim CTE that drops the shadowed piece (`WHEN next_start <= start_ts THEN NULL`), followed by a `FILTER (WHERE p2 IS NOT NULL)` in the downstream `merge()` aggregation. No shadowing was observed on the backfill of the 7 752 chains after the fix.

The second defect concerned the idempotence sentinel itself. The initial version applied `WHERE fused_at IS NULL` to identify the roots to fuse physically, which missed a case of practical importance: a chain that had already been physically fused but had gained new absorbed fragments at a subsequent cycle would not have been re-fused. On 857 absorbed fragments freshly created at one full cycle, 341, or close to 40%, pointed to a root whose `fused_at` was already non-null, and the canonical `tgeompoint` of those roots was not extended. The fix widens the sentinel by an explicit OR on chain growth (`fused_at IS NULL OR EXISTS (... AND a.merged_at > c.fused_at)`) that exploits the partial index `idx_stib_trip_cold_supby`. The cycle following the fix re-fused 294 roots in 0.7 s, and a third cycle confirmed the return to a strictly no-op state.

The third defect surfaced as a sustained hold on the advisory lock for more than ten minutes after an apparently successful run (`[fuse] 7752 ... 8.9s`). The cause was a final invariants check written as `NOT IN (SELECT trip_id FROM stib_trip_cold)` on some 565 000 rows. The PostgreSQL planner executes such an uncorrelated `NOT IN` as a hash anti-join over the full table, with no usable index, and three parallel workers remained blocked for the whole duration. The fix replaces the construct with a correlated `NOT EXISTS` scoped to the current run (`merged_at >= NOW() - INTERVAL '10 minutes'`), which uses the primary key and the existing indices. The post-fix measured cost is below 50 ms.

Limits of the present measurement and future work. The reported observation covers a single day of operation (nominal restart at 00:30 CEST followed by the offline backfill at 16:00 CEST). Three elements are missing for a fully controlled validation, and constitute the corresponding axes of Section 11. The first is the extension of the observation window to at least one week of continuous operation post-restart, a necessary condition to stabilise the fusion rate on the weekly cycle and to characterise the weekday-vs.-weekend-vs.-holiday modulation. The second is the constitution of a manually annotated sample of around fifty fusions, stratified by gap class (short versus long) and by distance, on which to estimate the chain-merge false-positive rate with a Wilson confidence interval, following the protocol of Section 9.4. The third is the execution of an iso-perimeter shadow run in which `hungarian_merge` and `greedy_chainmerge` operate in parallel on the same production window, a condition for a direct comparison on operational metrics.

The investigation of the eleven residual depth-3 cascade cases noted earlier is added to these three axes as a minor but measurable signal of a cross-cycle dynamic in the `superseded_by` forest.

7.6 The De Lijn Feed: Standard GTFS-Realtime

De Lijn publishes a standard GTFS-Realtime feed delivered as a JSON payload conforming to the GTFS-Realtime schema, exposed by De Lijn's Open Data API endpoint. For every sample, it already carries a stable `vehicle_id`, a `trip_id` matching the static schedule, and a latitude/-longitude pair. Transforming this feed is mostly deserialisation rather than reconstruction.

The De Lijn transformer decodes the message using the official GTFS-Realtime schema [4], joins each row against the static feed to recover trip metadata, converts WGS-84 to EPSG:3857 metres, and writes a normalised observation record. Defensive logic handles the usual edge cases: trips not yet in the static feed (relief services), missing or cold-start GPS positions, and duplicate snapshots.

Because De Lijn provides latitude/longitude, the along-line distance is computed post-hoc in the load stage by projecting onto the GTFS shape (Section 7.7). This is the reverse of STIB. STIB gives us the distance, De Lijn gives us the position, but after this step both feeds have the same shape, and downstream modules never branch on the operator.

7.6.1 Editorial Alerts: De Lijn GTFS-RT Alert Entities

Whereas STIB exposes its editorial alerts through a proprietary ODP stream, De Lijn publishes them in the standard GTFS-Realtime format, as `Alert` entities co-present in the same `FeedMessage` as the `VehiclePosition` entities. The De Lijn ingestor (`delijn_ingestor.py`) integrates these entities by piggybacking on the main position polling, without a separate cadence and without an additional HTTP request: the same response carries both object families, and the loader dispatches at the exit. This architectural asymmetry with STIB is imposed by the source formats: the GTFS-RT standard bundles alerts and positions, the STIB ODP format separates them.

Cadence and categorisation. Piggybacking on the 20-s position polling has two consequences. First, there is no separate 600-s cadence as on the STIB side: De Lijn alerts are refreshed at the rate of the main feed. Second, the persistence table receives writes at every cycle, which makes the `content_hash` deduplication particularly useful.

The table `rt.delijn_traveller_message` mirrors the general structure of its STIB counterpart, extended to absorb GTFS-RT specifics: `message_id` (primary key) is the `entity.id` of GTFS-RT, stable from one cycle to the next as long as the alert is published; `content_hash` carries no uniqueness constraint, unlike the STIB table (a same `entity.id` may see its content vary over time, a text update or a period extension feeding the life-cycle triple without triggering the creation of a new row); `cause` and `effect` are both `SMALLINT` integers of the GTFS-RT protobuf (for instance `cause = 9` for `CONSTRUCTION`, `cause = 6` for `ACCIDENT`, `effect = 1` for `NO_SERVICE`, `effect = 4` for `DETOUR`); `category` is a projection computed by the function `_derive_category(cause)` onto one of five labels {`other`, `technical`, `incident`, `weather`, `travaux`}.

The payload columns are the quadrilingual fields `header_fr/nl/en/de` and `text_fr/nl/en/de`; `route_ids[]` and `stop_ids[]` (`NOT NULL DEFAULT '{}'`); `active_start`, `active_end`, `active_periods` (GTFS-RT `TimeRange` intervals serialised in JSONB to absorb multiple active windows, `NOT NULL DEFAULT '[]'`); the lifecycle triple `first_seen_at`, `last_seen_at`, `resolved_at` identical to the STIB table; and `raw` (`JSONB NOT NULL DEFAULT '{}'`) preserving the original GTFS-RT payload.

The API endpoint `GET /api/delijn/traveller_info?active=1&limit=1000` exposes the active messages under the same convention as the STIB route.

The mapping `cause` → `category` performs a homogenisation of GTFS-RT labels onto the five common categories used by the frontend. Schematically: `CONSTRUCTION` → `travaux`, `ACCIDENT` → `incident`, `WEATHER` → `weather`, `TECHNICAL_PROBLEM` → `technical`, while `STRIKE`, `DEMONSTRATION`, `POLICE_ACTIVITY` and `MEDICAL_EMERGENCY` → `incident`; the remainder (`UNKNOWN_CAUSE`, `OTHER_CAUSE`, `HOLIDAY`) maps to `other`. This reduction onto five classes is motivated by the frontend's needs, in particular by the TTS trigger which fires only for `category == "travaux"`: the raw GTFS-RT granularity carries no application value beyond this partition.

Derivation of the De Lijn ***niveau****.* Unlike STIB, the GTFS-RT De Lijn feed carries no native priority level. The Flask service `delijn_service.py` projects the pair `(cause, effect)` onto an integer `niveau` $\in \{1, 2, 3\}$ by a cascade of conditions that exploits both the cause and the effect, and which integrates a temporal dependency for current works.

`niveau` 1 (high) fires if `effect` $\in$ {`NO_SERVICE`, `REDUCED_SERVICE`} or `cause` $\in$ {`STRIKE`, `DEMONSTRATION`, `ACCIDENT`, `POLICE_ACTIVITY`, `MEDICAL_EMERGENCY`}; it covers major service disruptions and serious external events.

`niveau` 2 (medium) fires if `effect` $\in$ {`SIGNIFICANT_DELAYS`, `DETOUR`, `STOP_MOVED`}, or if `cause` $\in$ {`CONSTRUCTION`, `MAINTENANCE`} and the current date in `Europe/Brussels` local time falls within the alert's active period (`active_start` not null and matching today); it covers significant slowdowns and works active on the day.

`niveau` 3 (low) fires for the rest, covering general information messages and works that are planned but not active today.

The logic exploits both `cause` and `effect`, in contrast to the `_derive_category(cause)` categorisation which relies on the cause only. This distinction enables the frontend to separate a severely disrupting event (`effect = NO_SERVICE`) from a merely planned roadwork, which the cause alone cannot do. The asymmetry of mechanism between the STIB projection and the De Lijn derivation justifies the term *homogeneous projection* used for the `niveau` field exposed to the frontend, rather than that of intrinsic feed property.

7.7 Trajectory Reconstruction by Shape Projection

Once a vehicle is matched to a trip and the trip has a GTFS shape, the platform builds its geographic trajectory by *linear referencing* onto that shape. The shape is a polyline with cumulative arc length from its start.

The platform stores, for every observation, the vehicle's position on the shape as a single scalar distance. Recovering the latitude/longitude at any point is a walk along the polyline until the requested distance is exhausted, with linear interpolation between the two surrounding vertices.

For STIB, the input is the raw along-line distance from the ODP feed, already measured from the line's start in the vehicle's current direction. For De Lijn, the WGS-84 position is projected onto the shape and the cumulative arc length of the projection foot is reported.

7.7.1 Why Linear Referencing Is the Right Abstraction

The scalar-distance representation is intrinsically ordered: two vehicles on the same shape can be compared by subtracting their along-line distances, yielding the true curvilinear gap. It is also robust to small perpendicular GPS noise, and compact to store. Its limit is that it only applies along the shape the vehicle is actually following. When the projection error grows beyond a few tens of metres (street closure, diversion), the platform leaves the along-line distance `NULL` and falls back to raw latitude/longitude for display.

7.7.2 Empirical Validation of the GTFS Shape Geometry

Linear referencing is only useful if the shape it refers to actually traces the road. The whole platform leans on PostGIS's `ST_LineLocatePoint` / `ST_LineInterpolatePoint` pair to translate between the scalar along-line distance and a geographic position; if the published GTFS shape drifts off the physical roadway, that translation places the bus a few metres into a building or a parallel street.

To quantify this risk, we measured the deviation of the published shape against three reference road networks of complementary nature: OSM (crowd-sourced, network-wide coverage), SMV (institutional, transit-priority subset), and UrbIS-Adm (photogrammetric, geometrically authoritative).

Method. For every line / direction shape in `static.stib_line_shape` (132 shapes after deduplication), the geometry is reprojected to EPSG:31370 (Lambert72, the official Belgian metric reference), and a regular sampling grid is generated along the polyline at a step of 5 m via $\texttt{ST\_LineInterpolatePoint}(g, i/N)$ with $N = \max(2, \lceil L_g/5 \rceil)$ to ensure at least two samples per shape, yielding 244,988 sample points total (171,341 bus, 51,407 tram, 22,240 metro). The Brussels-clipping mask is $\texttt{ST\_Buffer}(\texttt{ST\_ConcaveHull}(\texttt{ST\_Collect}(g), 0.85, \mathit{false}), 200)$, with target percent 0.85 chosen so the hull tracks the convex outer envelope of the SMV/UrbIS coverage without leaking into uncovered enclaves. Each sample point is then probed against three reference networks through a k-NN nearest-segment query (`ORDER BY geom <-> ref_geom LIMIT 1`):

- **OSM** [66] — the Brussels road and rail extract, 5,904 km / 57,460 ways. The exact filter is the negative `highway NOT IN {footway, cycleway, path, steps, bridleway, corridor, pedestrian, track, construction, proposed}` ∪ `railway IN {tram, subway, light_rail, funicular}` (the funicular branch picks up the *Coudenberg / Mont des Arts* CK4-Mont des Arts line without forcing a separate filter).
- **SMV** [67] — the strategic multimodal road classification published by Bruxelles Mobilité as part of the Good Move plan, 2,089 km / 12,898 features. The transit-priority subset (`tp_class` > 0) is reported separately as "SMV TP" (829 km / 3,741 features).
- **UrbIS-Adm** [68] — the cadastral-grade geographic reference of the Region of Brussels-Capital, with a planimetric accuracy of ∼25 cm, two orders of magnitude finer than the deviations measured here, which makes it suitable as ground truth. UrbIS encodes the roadway as 30.7 km^2 / 32,912 pavement *polygons*, so a sample point at distance 0 does not just sit close to the road centreline: it sits *inside the pavement polygon*.

The bus subset of the sampling, clipped to the Brussels-Capital mask to exclude legitimate extra-region terminus segments (lines 76/77 to Tervuren, line 12 to Brussels Airport), gives 162,839 STIB-bus points. Distances are then summarised per reference network.

A note on dimensional asymmetry between references. UrbIS encodes the roadway as 2-D polygons, so $\texttt{ST\_Distance}(\mathit{point}, \mathit{polygon}) = 0$ as soon as the point is inside any pavement of the road, regardless of its lateral position within the lane. OSM and SMV encode road *centrelines* as 1-D polylines, so a sample point cannot satisfy $\mathit{dist} = 0$ even on a perfect alignment: a vehicle in the right lane of a 2×2 boulevard is mathematically at ∼3.5 m of the OSM centreline. The median offset of ∼1.4 m on the OSM row (Table 7.9) therefore reflects approximately the half-width of typical Brussels lanes, not a STIB-shape error. Direct comparison of the UrbIS median (0 m) and the OSM median (1.43 m) is uninformative as a measure of relative precision; only the upper-tail percentiles can be compared meaningfully across the two encodings.

A note on partial circularity between references. SMV is derived from the regional UrbIS cadastre, and the STIB shape ingestion may itself rely on the same regional reference, so the three sources are not fully independent: the UrbIS and SMV cross-checks should be read as *consistency* indicators rather than as fully external references. OSM, sourced from a worldwide community map with no Brussels-specific dependency on UrbIS, is the only fully external cross-check in this comparison.

Result. Table 7.9 reports the bus-only / Brussels results against the three references. The headline number is on the UrbIS row: 96.94% of the 162,839 sampled points fall *inside* a UrbIS pavement polygon, with a Wilson 95% confidence interval of ±0.08 percentage points; the median and the 90$^{\text{th}}$ percentile are at 0 m, and the 99$^{\text{th}}$ percentile is below $\sim$ 273 m. Of the 3.06% outside any polygon, 4,226 points sit beyond the 10-m tolerance (the red markers of Figure 7.12), and a further 759 are strictly outside any polygon but within 10 m of one (off the polygon by less than a lane width, so not visually conspicuous).

Two distinct mechanisms account for the residual UrbIS gap, both *reference holes* rather than STIB-shape defects: (a) shape termini reaching beyond the regional border (lines 76, 77 towards Tervuren / Wezembeek-Oppem, line 12 towards Brussels Airport, where UrbIS does not cover the territory at all); and (b) tram corridors on dedicated rights-of-way through the Forêt de Soignes (lines 44, 39, 92), where UrbIS does not cadastre the vegetated tram platforms. Lines 42, 47, 50 and 74 trail in the 57–82% band and represent genuine shape-vs-physical-road divergences (likely outdated GTFS revisions or unmapped turning loops); they are listed in Appendix D for follow-up. The OSM row provides the only fully-independent cross-check: 99.94% of the same points fall within 10 m of an OSM way, with a maximum deviation of 35.7 m, supporting the conclusion that the published shape and the physical road are within sub-lane tolerance almost everywhere.

Table 7.9: Geometric validation of STIB GTFS shapes against three reference networks (probe date: 2026-05-04; repository commit `5e40b9d`; PostgreSQL 17.9 / PostGIS 3.5.2). "Inside polygon" is meaningful only for UrbIS, which encodes the roadway as 30.7 km^2 of pavement polygons. All distances are in metres. Sample size: 162,839 bus points, clipped to the Brussels-Capital concave-hull mask (`ST_ConcaveHull(..., 0.85)` buffered by 200 m).

Reference	median	p90	p99	max	% ≤ 10 m	inside polygon
UrbIS (gold)	0	0	272.84	1,103	97.40%	96.94%
OSM	1.43	2.76	5.34	35.66	99.94%	—
SMV total	2.15	10.6	218	1,081	88.13%	—
SMV TP only	2.34	16.1	276	1,169	83.98%	—

Interpretation. The validation underwrites four facts that the rest of the chapter relies on.

First, the deviation between the published STIB GTFS shape and the UrbIS roadway polygon, evaluated at sample points generated by `ST_LineInterpolatePoint`$(g, i/N)$, has a median of 0 m, meaning that more than half of the 162,839 sampled points lie strictly inside the gold-standard pavement polygon.

Second, the "few tens of metres" fallback threshold mentioned at the start of this subsection is empirically grounded: at the 99$^{\text{th}}$ percentile the deviation against OSM is 5.34 m, and the 35.66 m maximum sets the upper bound of where the platform might mis-render a vehicle under the strictest interpretation.

Third, the SMV / SMV TP rows should not be read as STIB-shape errors; they reflect that SMV is a *strategic* multimodal classification published as a sparse subset of the road network,

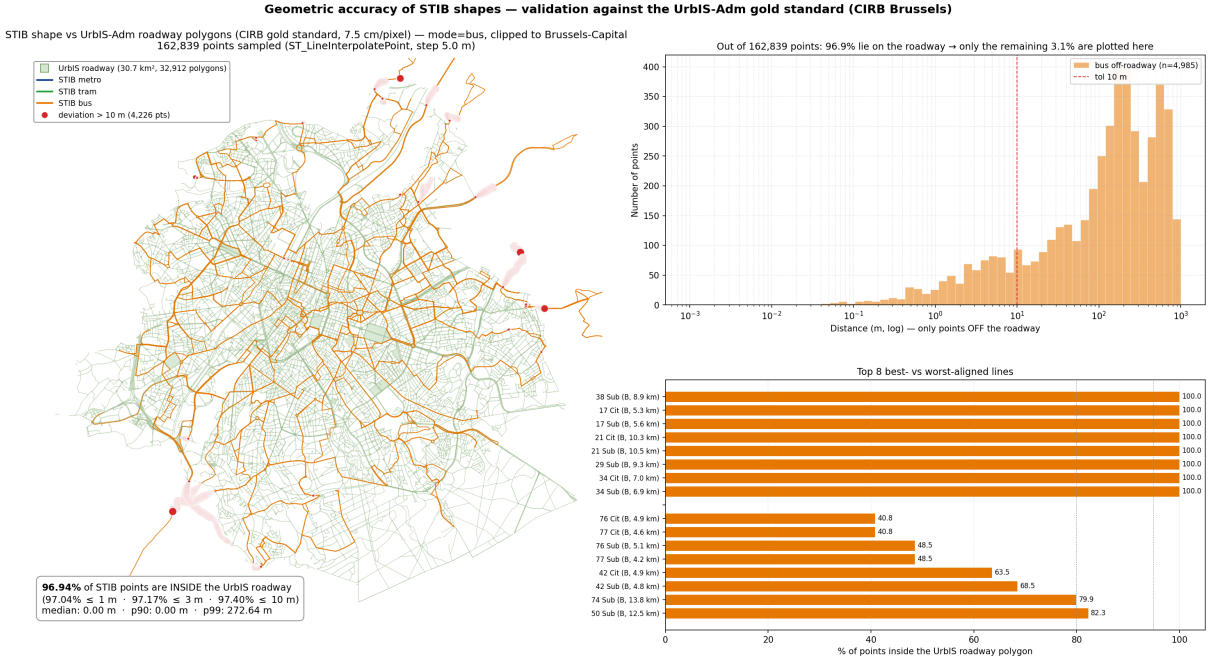


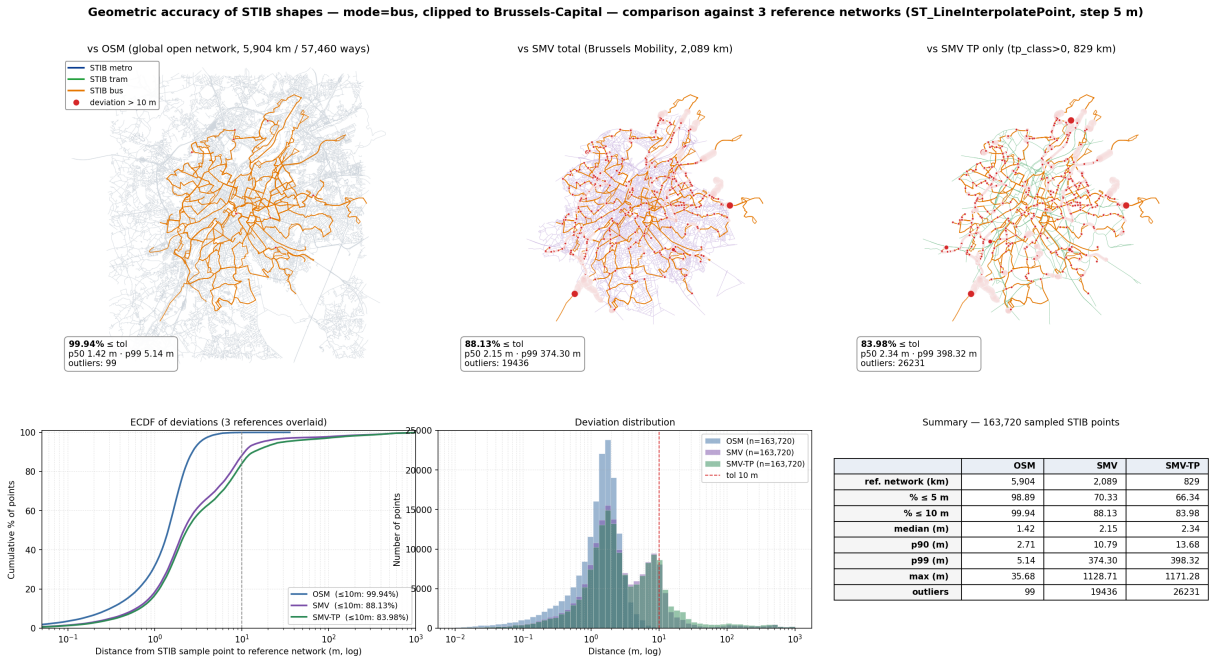
Figure 7.12: Geometric validation of STIB shapes against the UrbIS-Adm pavement polygons (CIRB, photogrammetric capture at 7.5 cm per pixel, ∼25 cm planimetric accuracy) on the bus subset clipped to the Brussels-Capital region. Left: STIB bus shapes (orange) overlaid on the UrbIS pavement polygons (green); the 4,226 outliers at $dist > 10$ m are marked in red (an additional 759 points are strictly outside any polygon but within the 10-m tolerance and are not displayed). The clusters reveal the two reference-hole mechanisms: extra-region termini (Tervuren, Wezembeek-Oppem, Brussels Airport) and tram corridors through the Forêt de Soignes. Top right: distribution of the off-polygon deviations on a logarithmic scale. Bottom right: per-line ranking, top eight best-aligned lines all at 100% inside polygon, bottom nine worst-aligned lines tracked for follow-up.

which is appropriate for Good Move analysis but not as a universal alignment reference.

Fourth, the small number of divergent lines in the 40–80% tail of the per-line bar of Figure 7.12 defines a tractable list of follow-up actions for the data publisher; they are propagated into the lines-coverage report of Appendix D for traceability.

The validation also exposes a measurement caveat that a careful jury will raise. `ST_LineInterpolatePoint` interpolates linearly between two consecutive shape vertices, so on tight curves (roundabouts, tram turning loops) the chord can step momentarily outside the pavement even when both bracketing vertices are inside it; the metric therefore measures, jointly, the alignment of the STIB shape and the density of its vertices. The effect is bounded because most STIB shapes carry a vertex every 5–10 m on curved sections, but it explains a non-zero fraction of the outermost tail of the deviation distribution.

A second concern, raised in defense rehearsals, is what happens to the bunching detector when a vehicle takes an unforeseen detour and the published shape no longer represents the physical road. Two mechanisms shield the detector against phantom bunching alerts in that case. First, the projection layer of §7.7 leaves the along-line distance at `NULL` as soon as the projection error grows beyond a few tens of metres, which removes the diverted vehicle from the along-line comparison that drives the proximity sub-detector (§7.11); the detector simply does not see that vehicle as a candidate for a bunching pair until it returns onto its planned shape.



	OSM	SMV	SMV-TP
ref. network (km)	5,904	2,089	829
% ≤ 5 m	98.89	70.33	66.34
% ≤ 10 m	99.94	88.13	83.98
median (m)	1.42	2.15	2.34
p90 (m)	2.71	10.79	13.68
p99 (m)	5.14	374.30	398.32
max (m)	35.68	1128.71	1171.28
outliers	99	19436	26231

Figure 7.13: Comparative geometric validation against the three reference networks (OSM, SMV total, SMV TP-only) on the same 162,839 bus sample points as Table 7.9. Top: STIB shapes overlaid on each reference, with off-tolerance points in red. Bottom: ECDF and histogram of deviations; OSM rises sharply to its 99.94%-at-10 m plateau, while SMV trails at 88.13% because the network is a strategic transit-priority subset, not a universal road map (cf. the partial-circularity disclaimer in §7.7.2). The summary table on the right collects the distributional statistics of Table 7.9.

Second, the freshness gate of §7.11.4 requires both vehicles in a candidate pair to have a recent observation within the per-line gate (180 s on STIB), so a stale projection cannot persist long enough to trigger a spurious alert. The two mechanisms compose: the deflected vehicle is silently dropped from the comparison rather than drifting through it.

7.7.3 Storage in MobilityDB

Both raw and linearly-referenced positions are stored as MobilityDB `tgeompoint` columns: a temporal sequence of geographic points with linear interpolation between instants [1]. A whole trajectory lives in one row, and MobilityDB's temporal algebra answers "where was this bus at 16:03?" or "how fast was it between 16:00 and 16:05?" without application-level code.

Trajectories are written incrementally on both feeds. The first observation of a new trip yields an `INSERT ...ON CONFLICT DO NOTHING` with a single-instant `tgeompoint`. Each subsequent observation triggers an `UPDATE` that calls `appendInstant()` (the procedural analogue of the `&+` operator [1]) on the existing `tgeompoint`. The mechanics are identical between feeds.

The only feed-level difference is the source of the trip key. De Lijn carries a stable `vehicle_id` in the GTFS-Realtime payload. STIB exposes no stable identifier, so the transformer's matcher computes a synthetic `vehicle_uuid` (SHA-256 over line, direction and scheduled start time) before the load layer runs. The in-database shape is a single `tgeompoint` per trip in both cases, so the downstream temporal algebra operates uniformly.

7.7.4 Benchmark: `stib_vehicle_position` (Flat) versus MobilityDB-Backed `stib_trip`

The platform keeps the raw flat table `rt.stib_vehicle_position`, one row per observation, with `fetched_at` and a `geometry(Point,4326)`, alongside the MobilityDB-backed `rt.stib_trip`, where each row is a full trajectory stored as a single `tgeompoint`. The coexistence is deliberate: it lets us measure, on the same live data, what MobilityDB buys us. Table 7.10 reports the measurement on the live production database (2026-04-23).

Metric	`stib_vehicle_position` (flat PostGIS)	`stib_trip` (MobilityDB)	**Ratio / note**
Row count (24h)	1 835 566 rows	68 114 trips	∼1 trip / 27 obs
Row count (all time)	13 501 554 rows	565 184 trips / 7.25 M instants	,
`pg_total_relation_size`	4 439 MB	2 757 MB	MobilityDB 38% smaller
`pg_relation_size` (data only)	1 818 MB	442 MB	MobilityDB 4.1× denser
`pg_indexes_size`	2 620 MB	2 255 MB	GiST on `tgeompoint`
Bytes/obs (data only)	141 B	63 B	2.2× denser
Bytes/obs (data + indexes)	344 B	390 B	equivalent
Q1: all positions of line 71, last 5 min	2.3 ms	8.2 ms	**flat wins** (B-tree)
Q2: position of bus X at a given instant	0.13 ms	4.2 ms (`valueAtTimestamp`)	**flat wins**
Q3: full trajectory of a run (playback)	0.12 ms	0.10 ms	draw
Q4: min-distance between two buses, 10-min window	impracticable	4.0 ms (`minDistance` on `atPeriod`-clipped pair)	MobilityDB-only

Table 7.10: Measured benchmark of the flat table `rt.stib_vehicle_position` versus the MobilityDB-backed `rt.stib_trip` on the live production database (PostgreSQL 17.9 / PostGIS 3.5.2 / MobilityDB 1.4.0, probe date 2026-04-23). Query times are the median of five warm runs (first run discarded) from `EXPLAIN (ANALYZE, BUFFERS)`. The exact SQL for Q1–Q4 is reproduced in Appendix B.

The direct conclusion is that on this workload *expressiveness, not speed, is the MobilityDB advantage*. The platform keeps both representations because the real operational queries are a mix. The public API answers its hot-path requests from the flat table and delegates to `stib_trip` only when the question is about a trajectory as a whole (the bunching detector, the ad-hoc analysis endpoints, the pre-seeded window of the QGIS columnar cache of Section 6). This is also why Section 8 argues the two tracks are complementary rather than competing.

7.8 Loader Refactoring: Hot/Cold Separation and SQL-Side Matching

The original STIB loader (called v1 here) reconstructs trips by iterating over a Python state dictionary of open trips. For each incoming GPS position, the loader computes a Haversine distance against every candidate, picks the closest, and emits an individual `UPDATE rt.stib_trip SET trip = appendInstant(...)`.

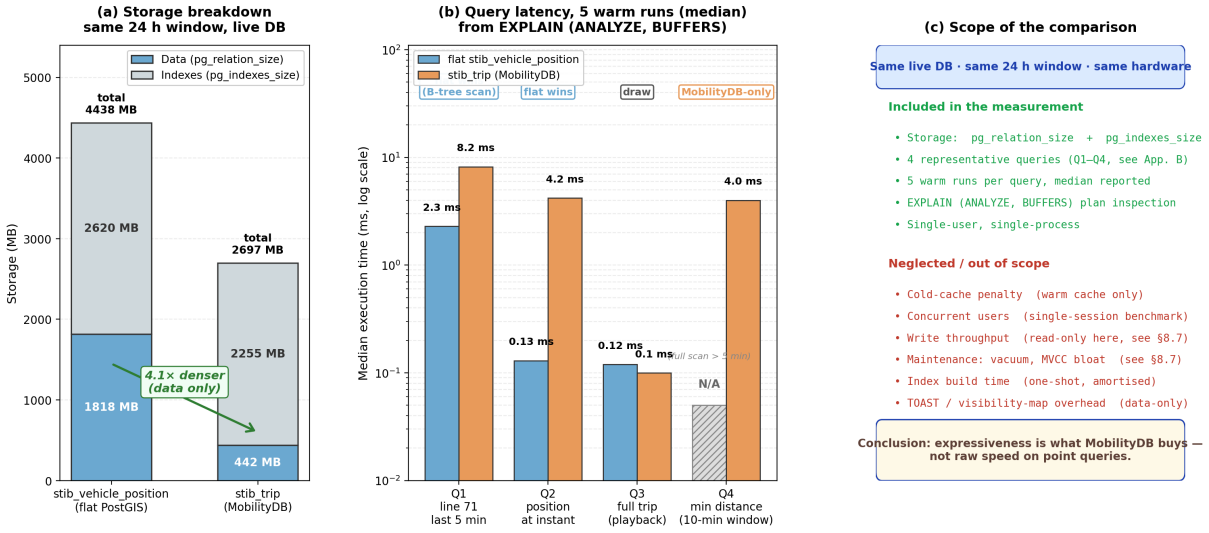


Figure 7.14: Visual summary of Table 7.10. **(a) Storage breakdown.** Stacked bars: data-only (`pg_relation_size`) plus indexes (`pg_indexes_size`); the totals match `pg_total_relation_size` from Table 7.10 within rounding. The 4.1× density advantage is on the data side only: once the GiST index on `tgeompoint` is included, the total-on-disk gap shrinks to ≈ 1.6×, which is why the platform keeps both representations rather than claiming wholesale storage savings. **(b) Query latency, log scale.** Median over five warm runs (first run discarded), from `EXPLAIN (ANALYZE, BUFFERS)` on the exact SQL of Appendix B. The flat Q4 bar is hatched ("N/A") because a full pairwise distance scan over a 10-min window did not return within five minutes; the bar is drawn only as a visual placeholder. Per-query badges record the dominant access path on each side. **(c) Scope of the comparison.** Methodology panel listing what the measurement includes (storage breakdown, four representative queries, warm-cache median over five runs, plan inspection, single-user) and what it on purpose excludes (cold-cache penalty, concurrent-user contention, write-side throughput, vacuum/Multi-Version Concurrency Control (MVCC) maintenance covered separately in §7.8, index-build amortisation, `TOAST`/visibility-map overhead). The conclusion is operational, not a global benchmark verdict: *expressiveness is what MOBDB buys, not raw speed on point queries.*

Two compounding costs become measurable as the network grows. A cycle of ∼1 200 positions issues ∼1 200 round-trips to PostgreSQL, each paying network, parsing and planning overhead. And under PostgreSQL's multi-version concurrency control (MVCC — the mechanism that keeps old row versions visible to in-flight transactions), every `UPDATE` creates a new tuple version and marks the old one as *dead*. On a long-trip `tgeompoint`, each call also rewrites a 30–50 KB TOAST blob. Dead tuples accumulate until `autovacuum` reclaims them, so `rt.stib_trip` grows to several gigabytes despite a modest live row count.

A subtler observation: `rt.stib_trip` already carried a GiST index on its `tgeompoint` column, but the matching logic never queried it. It preferred the in-memory state dictionary, so the existing spatial infrastructure was idle.

7.8.1 Design

A revised loader (v2) addresses both costs inherently along three axes.

Hot/cold separation. Open trips — the only rows receiving `UPDATE`s — live in a small dedicated table `rt.stib_trip_open` (∼2 000 rows in steady state). When a trip times out (no update

for 30 min), it is atomically graduated into the append-only archive `rt.stib_trip_cold`. A `rt.stib_trip UNION ALL` view exposes a single read interface to downstream consumers. The archive sees zero `UPDATE`s and is therefore bloat-free by design.

SQL-side spatial matching. A materialised `last_point geometry(Point,4326)` column on `stib_trip_open` carries a GiST index. The matcher issues a single `ST_DWithin` query that returns the candidate set already filtered by line, direction, time window and a coarse 500 m radius. The per-batch assignment of Section 7.4.1 (Hungarian over the at-most-tens of (*obs*, *open_trip*) pairs surviving the four hard rejects) then runs in Python on the prefiltered candidate set rather than on the full state dictionary. The optimisation therefore does not add infrastructure; it finally uses the spatial index the database already provided.

Bulk writes. The per-cycle `UPDATE`s and `INSERT`s are coalesced into one `UPDATE ...FROM (VALUES ...)` and one multi-row `INSERT`, dropping the round-trip count from ∼1 200 to roughly 10 per cycle.

The loader v2 architecture is orthogonal to the matching decision rule: the hot/cold split, the SQL-side prefilter, and the bulk writes all concern *access pattern*, while the per-line dynamic gates and the zero-risk merge of Section 7.4.1 concern *which trip an observation belongs to*. Both layers are deployed together; their effects are reported separately in Section 7.5 (matching quality) and in the present subsection (loader throughput). Figure 7.15 summarises the before/after architecture of the loader.

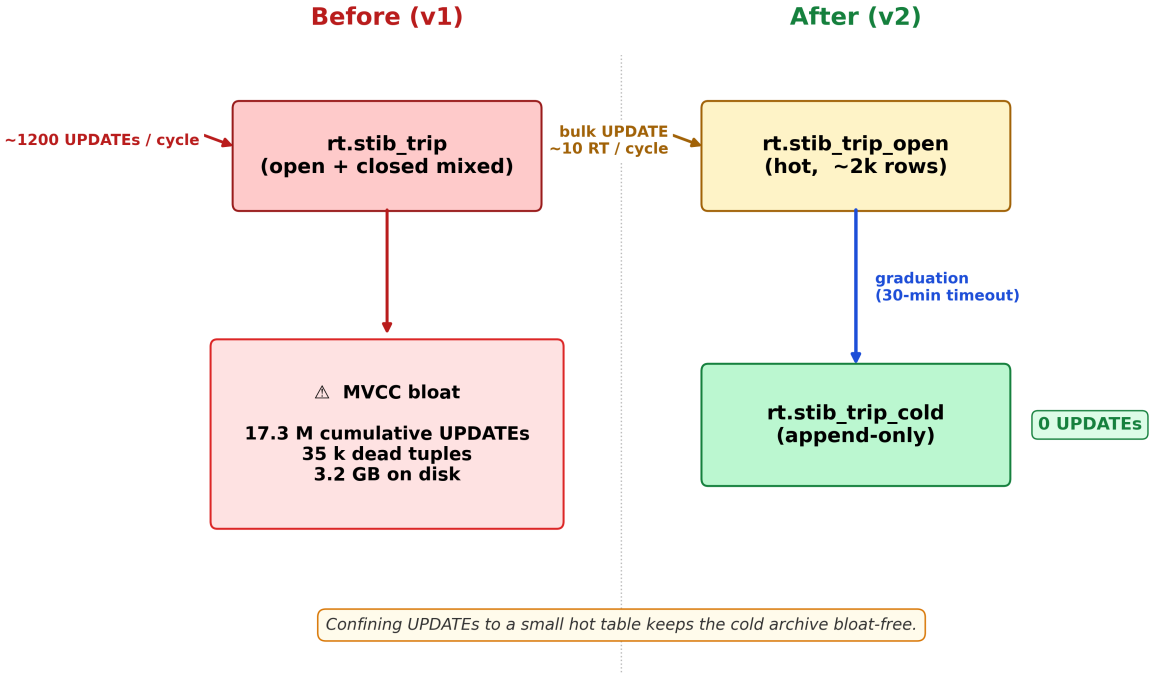


Figure 7.15: Hot/cold separation for the STIB loader. **v1:** a single table absorbs both the frequent updates of open trips and the long-term storage of closed trips, so MVCC dead tuples and TOAST rewrites accumulate on the whole table. **v2:** updates are confined to the small `rt.stib_trip_open` table; on timeout, a trip graduates atomically into the append-only archive `rt.stib_trip_cold`, which never sees an `UPDATE` and therefore stays free of MVCC bloat. An additional `rt.stib_trip UNION ALL` view (not shown) exposes both tables as a single read interface to downstream consumers.

7.8.2 Validation by Shadow Run

The two loaders were compared by a 25-hour *shadow run*. The refactored component runs in parallel with the baseline against identical input, so any output divergence comes from the change under study rather than from environmental variation. Concretely, an atomic tee in `stib_transform.py` writes each transformed batch to two staging directories. Then `load_stib.py` (v1, namespace `rt`) and `load_stib_v2.py` (v2, parallel namespace `rt_v2` used only for the bench) consume them sequentially in the same orchestrator. The `rt_v2` namespace was a temporary side-by-side workspace for the shadow comparison; once v2 was retained as the production loader, the hot and cold tables were re-unified under the standard `rt` schema (`rt.stib_trip_open`, `rt.stib_trip_cold`, view `rt.stib_trip`), which is the layout used throughout the rest of this chapter and in Appendix A.

Activation is gated by a marker file (`.bench_active`) and requires no restart. A shared `bench_logger` writes per-cycle CSV with identical columns, and an hourly cron snapshot records table sizes and `pg_stat_user_tables` counters. A cold-start seed copies `rt.stib_trip` rows with `end_ts > NOW() - 30 min` into `rt_v2.stib_trip_open` on activation, so v2 starts at functional parity with v1. The bench ran from 2026-04-23 20:47 UTC to 2026-04-24 22:00 UTC, processing ∼4.6 million GPS positions in each pipeline.

Two limitations of the protocol deserve mention. Both loaders share a single PostgreSQL instance, so a theoretical cross-effect via the common WAL and `fsync` budget cannot be ruled out. But v1 medians are stable before, during and after v2 activation, so the effect sits below the observed noise. And the cold-start seed reconstitutes only the persisted state of open trips. The Python-auxiliary variables (`last_dir_label`, `last_pos_m_on_shape`) are not copied, which accounts for part of the residual functional divergence reported below.

7.8.3 Results

Metric	**v1**	**v2**	**Gain**
Per-cycle medians (4 760 v1 cycles, 4 301 v2 cycles)			
Total cycle time	874 ms	331 ms	×2.6
Insert positions	382 ms	63 ms	×6.1
Update trips	194 ms	57 ms	×3.4
Matching	18 ms	51 ms	×0.36
DB round-trips per cycle	1 175	10	×117
Storage state at end of run			
Trip table total size	3 222 MB	82 MB cold + 10 MB hot	, and
Cumulative `UPDATE`s on archive	17.3 M	0	,
Dead tuples on archive	35 625	0	;

Table 7.11: STIB loader v1 vs. v2, 25-hour shadow run on identical input (2026-04-23 20:47 to 2026-04-24 22:00 UTC). Per-cycle figures are medians; storage figures are absolute end-of-run values from `pg_stat_user_tables`. The cumulative v1 figures include the trip table's pre-bench history. **Gain column:** values above 1× mark a speedup; values below 1× (*matching*, ×0.36) mark a slowdown on that step alone, absorbed many times over by the speedups on the other steps. The graduation step (1.2 vs. 6.9 ms median) is intentionally absent from the per-step breakdown: it is a new step in v2 with no v1 counterpart, and its cost is already included in the total cycle time.

The headline numbers are a 2.6× end-to-end speedup per cycle and a 117× reduction in

DB round-trips. The matching step alone is slightly slower in v2 (51 ms vs. 18 ms): the SQL plus `ST_DWithin` path carries more overhead than a pure-Python Haversine loop on the current candidate set. But this loss is absorbed many times over by the gain on bulk writes.

More importantly, the cold archive is fundamentally bloat-free: zero `UPDATE`s, zero dead tuples, and an on-disk footprint that scales with live data rather than with the number of position updates. Functional equivalence between the two reconstructions stays under the pre-agreed 5% threshold (4.7% on trip count, 3.3% on point count). The residual divergence comes from cold-start state and minor numerical differences between the geographic `ST_Distance` and the Python Haversine.

Two engineering lessons follow. On a write-heavy MobilityDB workload, isolating the small set of rows that actually receive `UPDATE`s into a dedicated hot table eliminates a class of MVCC bloat that no amount of autovacuum tuning resolves cheaply. And collapsing ∼1 200 single-row round-trips into ∼10 batched statements per cycle dominates any per-row algorithmic optimisation at this scale, a familiar conclusion in OLTP engineering, but easy to miss when the matching loop "feels fast" in Python. The same pattern is readily transferable to the De Lijn loader and is listed in Section 11 as such.

The end-to-end latencies reported in Section 7.13.1 (median 10.2 s for STIB, 11.3 s for De Lijn) are those of the v2 loader. Under v1 per-cycle medians, the equivalent figure at peak load would be of the order of 25 s, well above any reasonable "live" threshold.

7.9 Calendar-Aware Context Modelling

Service patterns vary across the week and year, so the detector tags every observation with a *calendar regime*. Each moment is mapped to one of five day-types: `WEEKDAY`, `WEDNESDAY` (Brussels schools let out earlier, distinct afternoon profile), `VACATION` (weekday inside a Belgian school-vacation window), `SATURDAY`, `SUNDAY_HOLIDAY`, and one of six time bands (night, morning peak, off-peak, evening peak, evening, late-evening). Belgian movable feasts (Easter Monday, Ascension, Whit Monday) are computed from Easter via Meeus's Gregorian algorithm [69]. School-vacation windows are encoded as a small static table per academic year.

7.10 Dynamic Threshold Computation

The bus bunching detector needs a *threshold*: a pair distance below which two buses on the same line are considered too close, and below which an alert should be raised. A single fixed threshold (say 100 m) fails in both directions. On a peak corridor with 900 m nominal spacing it is almost the definition of a bunch, and on a night line with 9 km spacing it is still nominal. The platform instead computes a threshold proportional to the *expected* vehicle spacing.

7.10.1 Frequency-Based Expected Spacing

The platform computes a threshold proportional to the *expected* vehicle spacing on the line, given the current `day_type` and `time_band`. The reasoning is straightforward. If a line has a planned frequency of f buses per hour at the current time, and those buses travel at an average speed of v kilometres per hour, then the expected separation between consecutive buses along the line is:

$$E = \frac{60}{f} \times v \times \frac{1000}{60} = \frac{v \cdot 1000}{f} \quad \text{(metres)} \tag{7.6}$$

Here f is in buses per hour and v in kilometres per hour; the factor $1000/60$ converts v to metres per minute, so E is in metres.

The detector uses three severity thresholds as fractions of E: `HIGH` at 5% (buses in practice on top of each other), `MEDIUM` at 12% (partial bunch, recoverable), `LOW` at 20% (spacing degrading). The relative ordering matters more than the exact fractions, which were fixed empirically on a few weeks of traces.

7.10.2 Clamping

Equation 7.6 can produce unreasonable thresholds at the extremes, below the GPS noise floor when E is small, larger than the entire line when E is very large, so the platform clamps the computed thresholds to the practical range $[20\text{ m}, 500\text{ m}]$.

7.10.3 Fallback Chain

When the frequency-based computation fails (missing GTFS entry, new line, unusual calendar corner), the detector falls back through a short chain: line-specific daily average, then global daily average, then hard-coded defaults (30 m / 80 m / 200 m for HIGH / MEDIUM / LOW). The fallback is rarely triggered in practice; it exists to keep the system resilient in the corners.

7.11 Automated Bus Bunching Alerts

With shape-projected positions and dynamic thresholds available, the bus bunching detector is the operational case study chosen to validate RQ3 (operational value on Brussels feeds). It serves as the concrete artefact through which the analytical contract of goal G5 (§7.2) is realised: a stream of machine-readable alert events the platform emits continuously on the live STIB and De Lijn feeds. This section surveys its logic.

7.11.1 Two-Pass Distance Computation

Every detection cycle (one per minute) the detector groups currently-open trips by (line, direction) and considers all pairs within each group. For each pair the distance is computed in two passes.

The first pass is a cheap Haversine between the two most recent WGS-84 positions. If this exceeds 600 m, the pair cannot be bunching and is skipped, sparing the overwhelming majority of pairs from any further work.

The second pass is the real metric. When both vehicles have valid along-line distances on the same shape and direction, the comparison uses the absolute difference of those distances, and the true curvilinear gap. When along-line distances are missing, the detector falls back to the Haversine distance, which slightly overestimates the gap on curved segments — a heuristic approximation accepted for its constant-time cost in the per-cycle detection loop.

Algorithm 7.5 formalises the per-cycle procedure.

7.11.2 Pairwise Alert Emission

Once all plausible pairs are scored, the detector emits one alert per pair that crosses a threshold. There is no sorting and no one-to-one constraint: a cluster of three bunched vehicles raises three pairwise alerts (A–B, A–C, B–C). The design favours recall. Clients can group alerts by shared `vehicle_uuid` for a compact display, but suppressing signals at emission would hide the true extent of denser bunches in the archive.

Algorithm 7.5: RTDataHub bus-bunching detection cycle (frequency-aware adaptive thresholds with $[20, 500]$ m clamp and freshness gate).

Input: set of currently open trips $\mathcal{O}$, current cycle time T.
Output: set of emitted alerts $\mathcal{A}$.

1. $\mathcal{A} \leftarrow \varnothing$.
2. For each group G of open trips on the same (line, direction):
   (a) For each unordered pair (τ_1, τ_2) in G:
      i. If $\text{age}(\tau_1) > 180$ s or $\text{age}(\tau_2) > 180$ s: **skip** (freshness gate).
      ii. Compute $d_h \leftarrow \text{haversine}(\text{last}(\tau_1), \text{last}(\tau_2))$.
      iii. If $d_h > 600$: **skip** (cheap reject).
      iv. If $\text{shape}(\tau_1) = \text{shape}(\tau_2)$: $d \leftarrow |\text{last\_s}(\tau_1) - \text{last\_s}(\tau_2)|$ (curvilinear); else $d \leftarrow d_h$ (fallback).
      v. Compute expected spacing $E \leftarrow f \cdot v$ (headway $\times$ cruise speed).
      vi. Set $\theta_H = \text{clamp}(0.05E, 20, 500)$, $\theta_M = \text{clamp}(0.12E, 20, 500)$, $\theta_L = \text{clamp}(0.20E, 20, 500)$.
      vii. If $d < \theta_H$: $s \leftarrow$ `HIGH`. Else if $d < \theta_M$: $s \leftarrow$ `MEDIUM`. Else if $d < \theta_L$: $s \leftarrow$ `LOW`. Else: **skip**.
      viii. Emit alert $(\tau_1, \tau_2, d, s, T)$ to `rt.<feed>_alert` and add it to $\mathcal{A}$.
3. **return** $\mathcal{A}$.

7.11.3 Alert Lifecycle

Each alert has a lifecycle: it opens when a pair crosses a threshold, it can escalate if the pair closes further, and it closes when the pair opens back up for more than a few minutes (with hysteresis to avoid flapping). Every open, escalate, and close transition is written to a persistent `alerts` table, which becomes a queryable archive of all observed bunching incidents over the life of the platform.

7.11.4 Freshness Gate on Paired Vehicles

One subtle trap in the design above surfaced in operational logs. The detector's in-memory state keeps a trip for `TRIP_TIMEOUT_S` = 1800 s after its last update, so transient ingestion gaps do not drop trips mid-journey. During that 30-minute grace window, a finished trip lingers in state with its terminus position frozen at its last fix. A naive pairwise comparison will then compare that frozen position against a live vehicle's current position, emitting a false HIGH alert whenever the terminus happens to sit within the Haversine prefilter window.

A concrete case observed on 2026-04-23 involved bus `B71_44`, whose trip ended at 13:15:24, being paired at 13:33:10 with `B71_29` at a reported distance of 73 m. The two positions were in fact 18 minutes apart, and only one of the two buses was actually on the map at the alert's timestamp. The display layers were internally consistent: the vehicle tile read `valueAtTimestamp` and correctly excluded `B71_44` at 13:33:10, and the alert tile read the `alerts` table and correctly showed the emitted alert. The inconsistency was purely on the detector side.

The fix is a second freshness gate, independent of `TRIP_TIMEOUT_S`. A batch reference timestamp t_{batch} is computed once per detection cycle as the maximum `last_seen` across open trips. Any pair whose either vehicle's `last_seen` is older than the per-feed threshold `PAIR_MAX_STALENESS_S` relative to t_{batch} is rejected before the Haversine prefilter runs. The

threshold is calibrated per feed rather than globally, because the two pipelines feed the detector with distributions of `last_seen` that differ in shape.

On the STIB side, `PAIR_MAX_STALENESS_S` = 180 s (`src/etl/pipeline/load/stib_alert_detector.py`). The position consumed by the detector is the output of the proximity matcher (Section 7.4.1), which projects an odometric position onto a reconstructed trajectory. ODP-side polling misses compound with the enrichment delay introduced by this projection step, which lengthens the tail of the `last_seen` distribution relative to the raw feed. The median sits at ≈20 s, the p_{95} at ≈81 s and the p_{99} at ≈282 s. The 180 s threshold sits between the p_{95} and the p_{99}, admitting most of the legitimate enrichment lag while rejecting outright-stale positions in the long tail that produced the original phantom alerts.

On the De Lijn side, `PAIR_MAX_STALENESS_S` = 60 s (`src/etl/pipeline/load/delijn_alert_detector.py`). The De Lijn feed delivers a direct GPS reading with a stable `vehicle_id`, with no intermediate odometric-projection step, so the intrinsic staleness is dominated by the upstream polling cadence alone (∼20 s plus ingestion jitter) and the `last_seen` distribution is tighter. The 60 s threshold matches three native polling cycles (3×20 s) and yields a gate strictly stricter than the STIB one.

The asymmetry 180 s versus 60 s therefore reflects two distinct structural properties: on De Lijn the threshold is calibrated on raw polling cycles; on STIB it is calibrated on the empirical post-enrichment distribution whose measured p_{99} justifies a more permissive bound. The detail of both detectors is reported in Appendix C. This calibration is feed-specific. An upstream change to the STIB API may shift the staleness distribution and force a periodic recalibration of `PAIR_MAX_STALENESS_S`.

The choice is also safe for the Haversine prefilter: a vehicle at 50 km/h covers ≈2.5 km in 180 s, well beyond the 600 m prefilter radius. So any pair the gate admits is one whose distance is still meaningful for the threshold rule. The 30-minute grace window is left unchanged, the new gate only guards emission, not retention. Past false-positive alerts already in the archive are kept as historical data about what the detector emitted at the time. Currently-open phantom alerts close naturally on the next detection cycle, because the resolver key they depended on is no longer produced.

7.11.5 Two Complementary Detectors: Slow Speed and Stuck

Two further detectors share the same architectural slot, the same freshness gate (180 s) and the same output table `rt.<feed>_alert`. The `alert_type` column tells them apart (see Appendix A). Their constants appear verbatim in Appendix C. We describe them briefly here. The limitations section (§7.15) lists them as detectors that we implemented but did *not* evaluate quantitatively in the present work.

slow_speed. For each open trip, the detector samples the position at T and $T - 120\,\text{s}$ and computes the haversine speed between the two samples. It then projects the position onto the active GTFS stop-segment to read the *planned* cruise speed for that segment, day-type and time-band. The ratio $r = v_{\text{obs}}/v_{\text{plan}}$ drives the severity cascade with thresholds tailored to slow-down rather than to inter-vehicle gap: $r < 20\%$ `HIGH` and $20\% \le r < 40\%$ `MEDIUM`. The cascade keeps a `low` branch in code for future tuning, but the SQL pre-filter at `SLOW_SPEED_RATIO_LOW` = 0.40 makes it unreachable in practice. These are looser than the 5%/12%/20% headway ladder used by the proximity detector because the two phenomena are different in nature: a bus moving at 30% of the planned cruise speed is a meaningful slow-down, whereas a bus sitting at 30% of the nominal headway distance is already a near miss. A sampling window (`SLOW_SPEED_SAMPLE_DT_S` = 120 s)

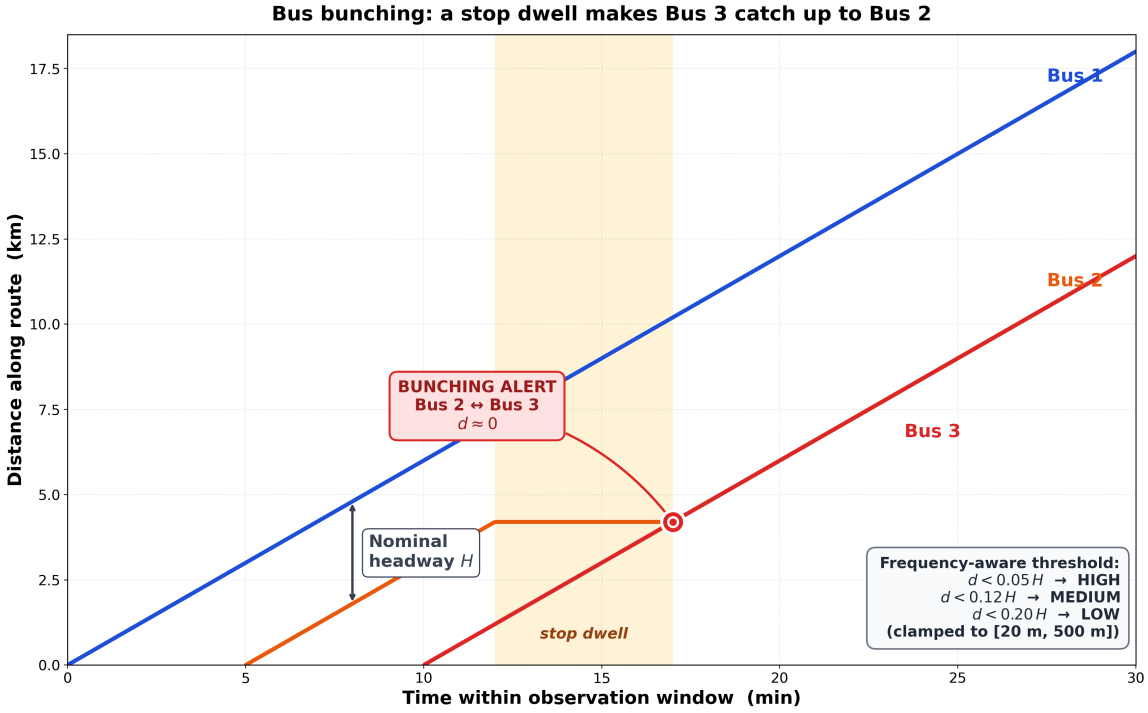


Figure 7.16: Bus bunching as a space-time phenomenon. The horizontal axis is time within an observation window, the vertical axis is distance along the route; each coloured line is the trajectory of one bus. Under nominal operation the buses maintain a steady headway distance H (illustrated between Bus 1 and Bus 2). At minute 12, Bus 2 stops at a stop for five minutes (passenger surge, traffic light, short turn): its trajectory flattens into a purely horizontal segment. Bus 3, which started one nominal headway later and travels at speed, catches up and the pair distance d collapses to zero, a *bunching event* (red marker). The detector of Section 7.11 raises an alert as soon as d falls below a fraction of the line-and-time-of-day-specific reference H (severity tiers and clamps shown in the legend box).

and a minimum inter-sample gap (`SLOW_SPEED_MIN_DT_S` = 10 s) keep brief traffic-light halts from triggering the detector.

stuck. The detector flags trips whose position has not advanced beyond a 20 m radius across a three-minute look-back, with at least three non-null samples in the window. It excludes the 80 m vicinity of any GTFS stop and the terminus zone around the trip's first or last stop. Severity escalates with the time spent without movement: `LOW` on first emission, `MEDIUM` after 5 min (`STUCK_SEV_MEDIUM_S` = 300), `HIGH` after 15 min (`STUCK_SEV_HIGH_S` = 900). The reasoning: multi-minute stalls in mid-corridor differ operationally from brief layovers.

7.12 API and Frontend

7.12.1 API endpoints overview

The API is a small Flask [15] application that exposes four groups of endpoints, each tied to a different concern.

The first group returns **vector tiles**. Every static layer (line shapes, stop markers) and every fast-changing layer (current vehicle positions, active algorithmic alerts) is served as MVT (Mapbox Vector Tile) via PostGIS's `ST_AsMVT` function. The SQL behind each endpoint clips the relevant table to the tile's bounding box, re-projects to the tile's local coordinate system,

and emits the resulting features in the binary tile format. The browser caches each tile at the `(z, x, y)` level and re-requests a tile only when it leaves the viewport and comes back in.

The second group returns **JSON metadata**: legends, line colours, the list of currently open algorithmic alerts, a per-trip trajectory for playback, and the usual administrative bits (the set of available lines, the current `day_type`, and so on). These pieces do not fit naturally into a tile, either because they are not geographic or because they need to be fetched in one piece.

The third group handles **temporal playback**. When the user moves the time slider, the browser requests the positions of all vehicles at the requested instant. The server evaluates this against the `tgeompoint` columns using MobilityDB's `valueAtTimestamp` function, which returns interpolated positions at the exact requested moment. A small amount of special-case logic handles Noctis night services: their trips span midnight, so their "operational day" needs a 3.5-hour offset.

The fourth group covers the **editorial Traveller Info** streams introduced in Sections 7.4.4 and 7.6.1: the endpoints `/api/stib/traveller_info` and `/api/delijn/traveller_info` expose the active editorial messages published by the operators and are queried by the floating P1/P2 alert bell and by the dedicated map layer described further below.

7.12.2 Frontend Architecture

The browser frontend is minimal: a single HTML page, MapLibre GL JS [6] for rendering, and roughly a thousand lines of plain JavaScript. No bundler, no transpilation, no framework: easy to read, easy to deploy. The flat JavaScript organisation by functional layer keeps the rendering code legible and avoids the overhead of a virtual re-render cycle that would be a poor fit for the high-frequency feature-state updates this application performs. The base layer is provided by MapTiler, which removes the need to self-host a base-tile instance.

This minimal architecture supports several mechanisms described in the following subsections: the strict layer-stacking invariant, the interactive subfilters and editorial alert bell, the Traveller Info map layer, the unified alert tooltips, the audio cues for P1/P2 alerts, the composite anomaly colouring of segment metrics, and the client-side timing constants.

The map composes a MapTiler OpenStreetMap-derived base with a stack of MVT layers from the Flask API. A control panel toggles layers, plays and pauses the temporal animation, and scrubs the time slider. Alert markers render on top of vehicle markers with a distinct style and a popup, the structure of which is detailed in Section 7.12.6 below.

7.12.3 Layer Stacking Order

The layout of the transport layers obeys a strict invariant organised in five conceptual tiers whose relative order must be preserved even when the rendering engine performs a style reload. For this reason, the order is re-imposed at every layer addition, which makes the invariant robust to hot-reloads and to dynamic style mutations. Within the top tier, a sub-order is also imposed between the alert sources: De Lijn pastilles sit at the bottom, STIB pastilles above, and TransportRT pastilles at the top, so that the most operationally pressing alerts remain pointer-accessible. The editorial Traveller Information layer (Section 7.12.5) is positioned below the algorithmic alert pastilles inside the alert tier, by a routine `moveBehindAlerts()` that prevents the editorial pictograms from masking the algorithmic markers (`proximity`, `slow_speed`, `stuck`) of greater operational urgency.

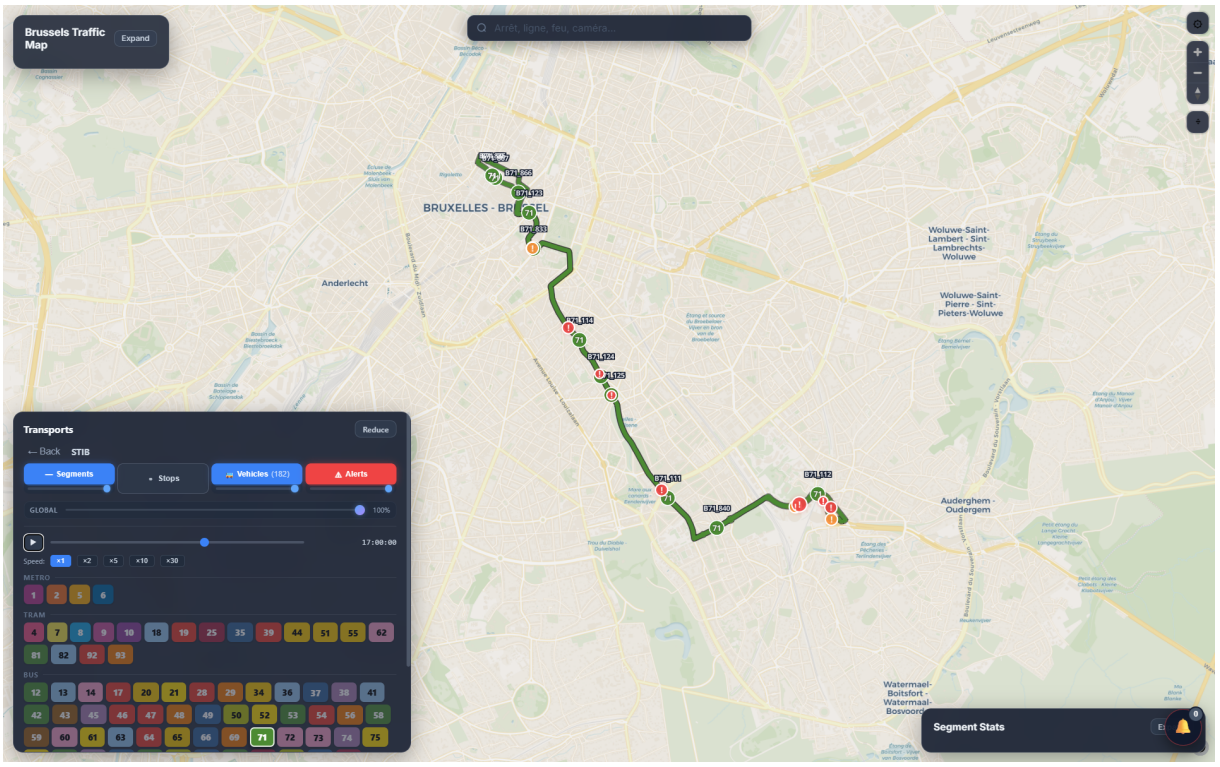


Figure 7.17: Live RTDataHub frontend on Saturday 2026-05-23 at 17:00:00 (evening-peak band, 16–19), with the STIB layer expanded in the *Transports* panel and line 71 selected (highlighted in the bus catalogue at the bottom). The map shows the network-wide STIB fleet of 182 vehicles; line-71 vehicles carry `B71_xxx` labels and are visible from the Bruxelles centre down to Auderghem-Oudergem and Watermael-Boitsfort. Six active bunching alerts appear along the corridor: one orange marker near `B71_833` in the Pentagon, two red markers around `B71_114` and `B71_125` on the Avenue Louise axis, and a cluster of three red and orange markers near `B71_111`, `B71_840` and `B71_112` close to the Auderghem terminus. The collapsed *Brussels Traffic Map* panel (top-left) carries the time-window controls and tile-layer toggles (TomTom flow / incidents, Waze jams); the collapsed *Segment Stats* panel (bottom-right) hosts the time-series explorer.

Table 7.12: Five-tier layer stacking order enforced at every layer addition. Within the alert tier (`t-alerts`), the sub-order is editorial Traveller Info < De Lijn algorithmic < STIB algorithmic < unified TransportRT.

Tier	Rationale
`t-casing`	Lower-edge band of centrelines (halo); must sit below the main stroke to produce the casing effect.
`t-lines`	Centerlines and alert-bearing segments; main network stroke.
`t-stops`	Stops and editorial information anchors; tied to the network, must remain readable above the lines.
`t-vehicles`	Vehicle markers; drawn above the network to support live-flow tracking.
`t-alerts`	Clickable alert pastilles; placed at the top to keep them pointer-accessible above vehicles. Internal sub-order: editorial Traveller Info < De Lijn algorithmic < STIB algorithmic < unified TransportRT.

7.12.4 Interactive Subfilters and Editorial Alert Bell

The TransportRT layer, which aggregates the operational real-time alerts, carries a subfilter panel collapsed by default and revealed by a disclosure control. Once expanded, the panel exposes

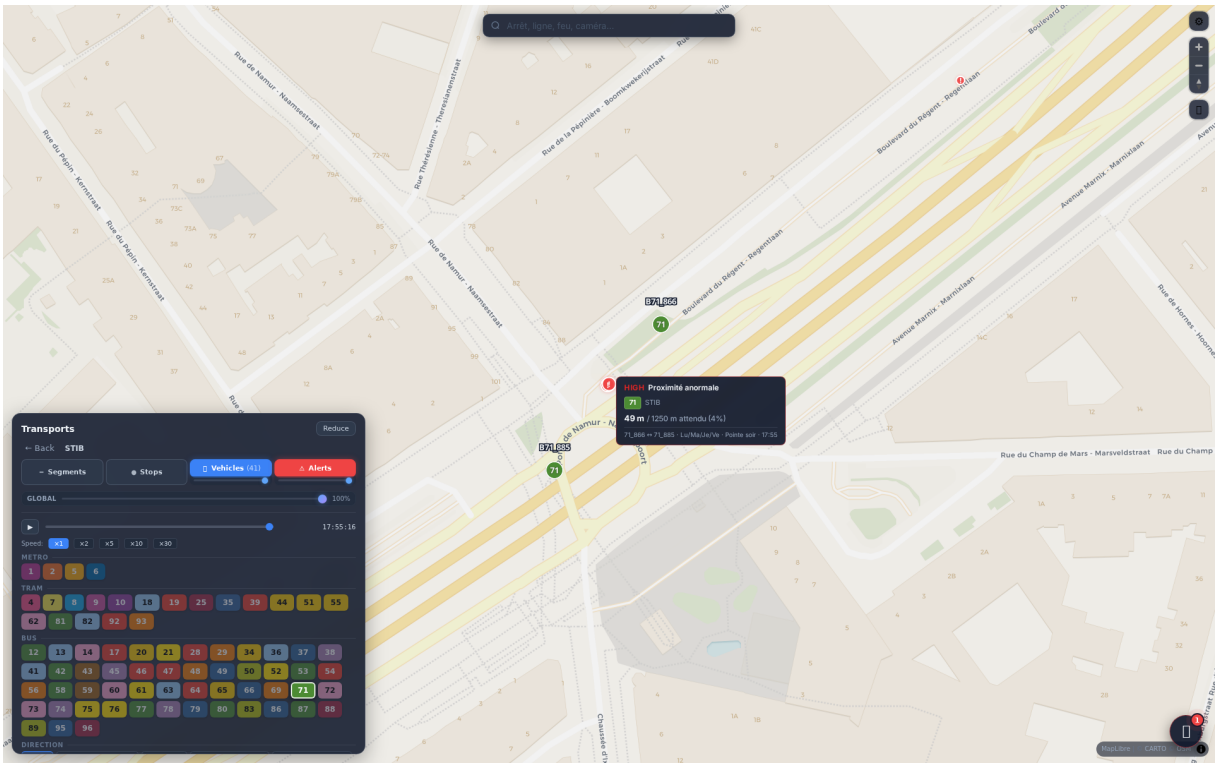


Figure 7.18: Close-up of a HIGH proximity (bunching) alert on the RTDataHub frontend, along the central line-71 corridor at Boulevard du Régent / Rue de Namur (Friday 2026-05-22, 17:55:16, evening peak). Two line-71 vehicles, `B71_866` and `B71_885` (visible as labelled green markers framing the red alert diamond), sit 49 m apart against an expected pair spacing of 1 250 m at this time of day, about 4% of the frequency-aware threshold, hence the HIGH severity. The popup, anchored on the alert marker, exposes severity, line, measured distance versus expected spacing, calendar context (`weekday` · `evening_peak`), the canonical pair (`71_866` ↔ `71_885`), and the detection timestamp. The *Brussels Traffic Map* (top-left) and *Segment Stats* (bottom-right) panels are hidden to give the alert visual room; the *Transports* panel (bottom-left) is left visible to make the layer activation chain explicit: the STIB drill-down, then line 71 selected in the catalogue, then the `Vehicles` and `Alerts` toggles active.

five functional controls: a multi-select priority group (P1, P2, P3), a *Sur Geolink* pill which restricts the display to alerts within thirty metres of a network segment, a *Nouveau aujourd'hui* pill ticked by default which retains only messages that first appeared today, a *Bus* pill which hides tram and metro services, and a *Mute* pill which disables the speech synthesis described in Section 7.12.7. Each change of pill state triggers an immediate client-side re-filtering without a fresh backend request.

The uniform interface masks an actual heterogeneity of the upstream feeds. On the STIB side, the raw `priority` field of the Open Data Portal typically takes values one through nine and is discretised to three levels by the Flask service `stib_service.py` through a tripartite `CASE WHEN` expression (1–3 → level 1, 4–6 → level 2, 7–9 → level 3; see Section 7.4.4). On the De Lijn side, the GTFS-RT feed carries no native priority level, and the service `delijn_service.py` derives the level by conditional cascade on the `(cause, effect)` pair (see Section 7.6.1). The frontend therefore receives two streams whose classification semantics differ, but whose final presentation is homogenised onto the same UI surface. By analogy, the *Bus* filter rests on a boolean `is_bus` field derived server-side from the GTFS `route_type`, where standard code 3

designates a bus service.

Independently of the TransportRT panel, a floating bell is anchored at the bottom right of the interface and signals on a permanent basis the P1 and P2 editorial alerts. The decoupling is intentional: the bell forms a permanent editorial alert channel that must remain visible even when the TransportRT layer is off, so that a user focused on another layer does not miss a major event. The bell runs its own refresh cycle on a sixty-second cadence, independent of the operational alerts, and queries the two editorial endpoints in parallel (`/api/stib/traveller_info` and `/api/delijn/traveller_info`) with the parameters `active=1&limit=1000` to aggregate the still-active messages from both operators. A strict filter on level retains only values one and two. The bell respects the *Sur Geolink* and *Nouveau aujourd'hui* pills but deliberately ignores the *Bus* and priority pills, since it is itself a closed P1/P2 filter. A pulsing animation with a 1.6-second period draws attention when the alert count is non-zero.

Table 7.13: The six subfilter pills of the TransportRT panel, grouped into five functional controls (the three priority pills form one multi-select group).

Pill	**Default**	**Filter logic**
P1	unchecked	Retain alerts whose homogenised `niveau` equals 1.
P2	unchecked	Retain alerts whose homogenised `niveau` equals 2.
P3	unchecked	Retain alerts whose homogenised `niveau` equals 3.
Sur Geolink	unchecked	Retain alerts within thirty metres of a Geolink segment.
Nouveau aujourd'hui	checked	Retain alerts whose first appearance falls today.
Bus	unchecked	Hide alerts for which the boolean `is_bus` is false (tram and metro).

7.12.5 Traveller Information Layer (Editorial Alerts)

Beyond the floating bell, the editorial messages are rendered as a map layer in their own right, accessible under the same TransportRT toggle as the algorithmic alerts but anchored on stops rather than on vehicles. This layer composes ten joint mechanisms which, together, make the editorial information legible at three scales of use: regional (Brussels-wide), neighbourhood, and stop-level.

Dedicated pictogram and progressive clustering. The pictogram used for editorial messages is a cyan-blue "i", chosen to stand apart from the red, orange and purple markers of the three algorithmic detectors (`proximity`, `slow_speed`, `stuck`). The visual semiology separates the two registries: continuous algorithmic computation on one side, operator-issued editorial announcement on the other.

With the production ratio of $\sim$2 impacted stops per message (467 active messages produce typically 1 090 markers), a naive rendering would saturate the regional view; the layer therefore relies on MapLibre's native progressive clustering, configured with `cluster: true`, `clusterRadius: 50` and `clusterMaxZoom: 14`. The regional view exposes a few large clusters carrying an abbreviated counter, the intermediate view decomposes them progressively, and the close view reveals the individual markers stop by stop. A click on a cluster triggers `getClusterExpansionZoom()` followed by an `easeTo()` that brings the map to the zoom level at which the cluster separates, turning the disaggregation gesture into a predictable animation.

Cluster priority propagation and graded stroke. A cluster carries more than its counter: the cluster property `min_priority` is set by the MapLibre aggregate `["min", ["coalesce", ["get", "min_priority"], 9]]`, which lifts to the cluster the urgency of its most pressing message. This information is made legible by a graded stroke applied identically to the individual markers and

to the clusters: thick red `#ef4444` (3 px on clusters, 2.5 px on markers) for `min_priority` ≤ 4, orange `#fb923c` (2 px) for values 5 and 6, thin white (1.5 px) for 7, 8 and 9. The direct and intended consequence: a cluster that contains a single message of priority 3 stands out in thick red even if it aggregates fifty messages of priority 9. The cyan fill remains the visual thread of the editorial layer; the stroke carries the severity.

Z-order placement below algorithmic alerts. Within the `t-alerts` tier of Section 7.12.3, the Traveller Info layers are deliberately placed below the algorithmic layers by a routine `moveBehindAlerts()`. The sub-order is consistent with the operational hierarchy: a vehicle currently stuck or in a bunching event is a more urgent event than an announced work site, and it matters that the algorithmic pastilles remain pointer-accessible even where multiple editorial messages accumulate.

Popup with language selector and coloured line chips. A click on an individual marker opens a MapLibre popup organised around the content of the impacted stop. The header lists the stop name (FR with fallback to NL and then to the identifier if the translation is missing), the number of attached messages, and a three-button language selector (FR, NL, EN) whose active button is cyan; the chosen language persists across subsequent popups within the same session. The body contains one card per message attached to the stop, structured around a priority badge (P3 to P9) with a left-graded border (red, orange, cyan as a function of the value), coloured line chips that exploit the official `route_color` and `route_text_color` read from `/api/stib/lines`, and the message text in the selected language with automatic fallback to the other two if the translation is absent.

Auto-refresh and line-selection watcher. While the TransportRT toggle is active, the layer triggers an automatic refresh every five minutes, a cadence consistent with the ten-minute ingestion cycle that avoids both network overload and visual obsolescence. A 1.5-second watcher monitors the `stibTp.selected` property of the line selection panel in parallel: whenever the selection changes, a new `fetch` is emitted with the parameter `lines=<csv>` that restricts the request to the actually selected lines, which avoids loading out-of-scope messages.

Asymmetric operator filtering. The layer shares with the algorithmic pastilles an operator filtering rule designed to remain intuitive in the presence of partial selections. When the user has selected no line on either operator, both endpoints are queried. When the user has selected at least one STIB line without selecting any De Lijn line, the STIB endpoint is queried with the line filter and the De Lijn endpoint is skipped altogether. The symmetric behaviour applies to a De Lijn-only selection. When both operators carry selections, each is queried with its own list. The asymmetry is intentional: it rules out the counter-intuitive outcome where a single-operator selection would cause the entire alert set of the other operator to appear.

7.12.6 Unified Alert Tooltips

The algorithmic detectors (`proximity`, `slow_speed`, `stuck`) are unchanged in the backend with respect to the description of Section 7.11; on the frontend side, however, their rendering has been factored into a presentation that is now shared across the three tooltip sites: the TransportRT IIFE (main map), the STIB panel and the De Lijn panel. The factorisation rests on three helpers reused identically across the three sites.

Helper `vehicleNoun(op, key)`. The first helper produces dynamically the vehicle noun to insert in the alert label. It looks up the appropriate catalogue (`stibTp.catalog` or `dlTp.catalog`) to obtain the mode of transport for the line at hand, then maps that mode onto a French noun. A tram line renders *Tram bloqué*, a metro line renders *Métro bloqué*, a bus line renders *Bus*

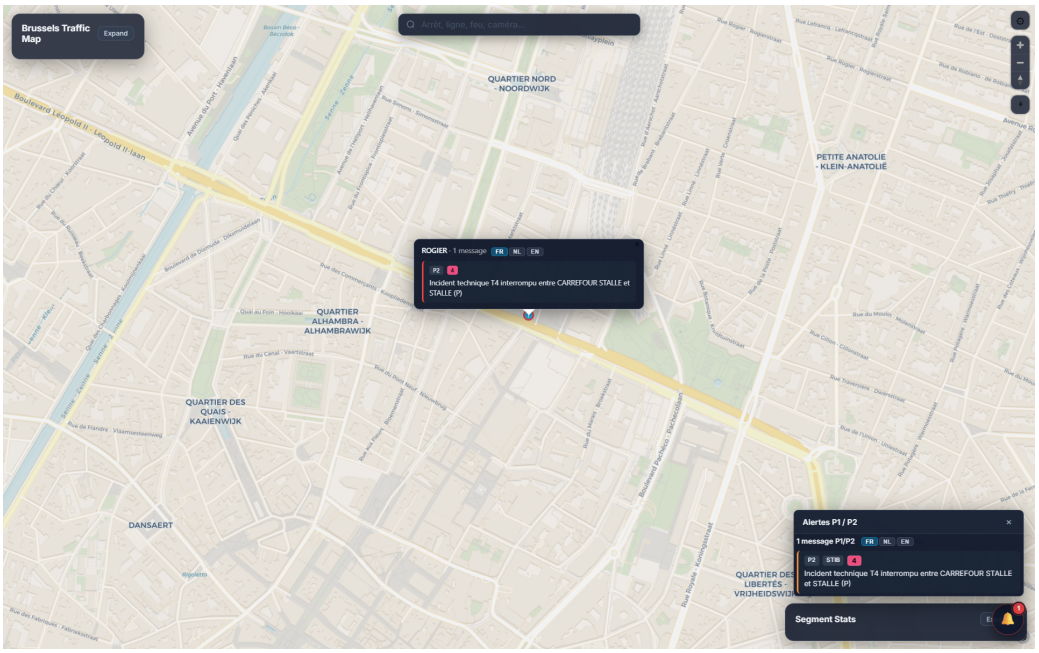


Figure 7.19: Floating P1/P2 editorial alert bell with its panel expanded. At capture time only one STIB P2 message is active (technical incident on tram T4 between Carrefour Stalle and Stalle (P), raw priority 4); the same message appears both on the in-map popup (centre) and in the bell panel (bottom-right). The bell aggregates STIB and De Lijn editorial messages of priority levels 1 and 2 under a single panel; the De Lijn side is empty at capture time.

bloqué, a rail line renders *Train bloqué*, and a funicular renders *Funiculaire bloqué*. The generic fallback *Véhicule* covers the rare cases where the mode is absent from the catalogue. The popup of segment alerts on the Transport side panel receives the same treatment, with a direct read on `props.line_mode` already exposed in the MVT tiles.

Table 7.14: Mapping from the line mode (catalogue field) to the French noun used in the alert label.

Mode (catalogue)	Noun rendered	Example for `stuck`
`metro`	Métro	Métro bloqué
`tram`	Tram	Tram bloqué
`bus`	Bus	Bus bloqué
`rail`	Train	Train bloqué
`funicular`	Funiculaire	Funiculaire bloqué
unknown	Véhicule	Véhicule bloqué

Helper `operatorLineBadge(op, key)`. The second helper produces the coloured chip that materialises the line identifier in the tooltip. It looks up the operator catalogue and reads the official colours `route_color` (fill) and `route_text_color` (text) when available. The rendering thus reuses the colour code established by STIB and by De Lijn in their official visual materials, so that the identifier of metro line 1 carries its characteristic burgundy, that of tram line 92 its proper blue, and so on. A grey fallback `#374151` is used for lines missing from the catalogue (in particular the STIB night buses N12 and N18 which do not appear in `/api/stib/lines`), so a chip is always rendered even on an incomplete catalogue.

Helper `renderAlertTooltipHtml(p, op)` four-section structure. The third helper assembles

the complete tooltip in a four-section structure that is strictly identical across the three sites. The first section is a header carrying a severity badge (`LOW`, `MEDIUM`, `HIGH`) followed by the alert-type label built with the dynamic vehicle noun of the first helper. The second section carries the coloured line chip of the second helper alongside the explicit operator name ("STIB" or "De Lijn").

The third section exposes the main metric, the form of which depends on the alert type: a `stuck` alert displays a phrase of the type *Immobile depuis X min* whose duration is formatted by the helper `_humanDuration`; a `proximity` alert displays a measure-expected-percentage triple of the form *45 m / 200 m attendu (22 %)*; a `slow_speed` alert uses the same structure in kilometres per hour, e.g. *12 km/h / 25 km/h attendu (48 %)*.

The fourth section, compact and rendered in grey with a top separator, aggregates the contextual secondary information: trip or vehicle identifier, segment, validity time band, and detection time at format HH:MM. The popup of the Transport side panel, dedicated to alert segments, receives the same structure and covers the three alert subtypes (`proximity`, `slow_speed`, `stuck`).

7.12.7 Audio Cues for P1/P2 Travaux Alerts

The editorial alerts trigger an audio notification by default, the user retaining the option to disable that modality (opt-out rather than opt-in). The trigger criterion is, however, restrictive by design: only the alerts whose level is one or two, whose category is `travaux`, and which have not yet been spoken during the current session are vocalised. The anti-repetition control rests on a client-side identifier journal capped at five hundred entries, which is enough to cover a typical session without saturating the browser memory. The choice to exclude level three and the categories other than `travaux` is meant to avoid acoustic saturation on transient incidents: punctual breakdowns, minor delays and weather phenomena are by nature unpredictable, whereas planned worksites form a calibrated editorial message whose audio announcement reinforces the informative value.

The vocalisation is performed by the browser Web Speech API, with a language fixed to `fr-FR` on the STIB side; on the De Lijn side, the language travels with the message itself according to the bilingual region served. The mute state is persisted locally so the user preference survives reloads; activating the mute also immediately interrupts any reading in progress.

7.12.8 Composite Anomaly Colouring of Geolink Segments

The rendering of segment anomalies on Geolink segments rests on the superposition of three distinct MapLibre layers, each dedicated to one of three per-segment anomaly counts: `proximity_count` for proximity density, `slow_speed_count` for passages at abnormally low speed, and `stuck_count` for prolonged-stop episodes. Each layer applies a linear interpolation across five colour stops, from cyan (absence of event) to red (saturation) through lime, yellow and orange.

The composite score displayed on each segment, `congestion_score`, is computed client-side as a Noisy-OR aggregation of the active sub-metrics (`proximity_count`, `slow_speed_count` and `stuck_count`). Each sub-metric is first normalised to $[0, 1]$ against its own historical baseline on that segment: $s_m = \text{clamp}((c_m - P_{75})/(P_{95} - P_{75}), 0, 1)$, where P_{75} and P_{95} are per-fid percentiles loaded once at startup from a precomputed `score_baseline.json` (24-hour rolling window, refreshed nightly). Segments with fewer than ten historical observations fall back to

global percentiles; per-metric minimum-count gates (3 for proximity and slow speed, 1 for stuck) silence the noise floor.

The composite is then $score = 1 - \prod_m (1 - s_m)$ over the active metrics, which by construction stays bounded in $[0, 1]$ and emphasises segments that flag on several axes at once. The score drives the colour interpolation across the five stops of Table 7.15 (cyan to red).

The per-fid baseline avoids the easy red artefact that a single global P_{95} would produce: the global P_{95} of `proximity_count` sits around 8 across the whole network, but reaches ≈60 on the busiest corridors of Avenue Louise or Chaussée de Wavre; without per-fid normalisation those corridors would saturate at red whether they are currently anomalous or merely busy.

A *time scrubber* is in addition provided for the segment alerts: it takes the form of a temporal slider that lets the observer navigate anomalies window by window, relying on a client-side `setFeatureState` override (the MapLibre feature-state API) with a graceful fallback to the pre-computed tiles when the override is not available. The context of use is discussed in Section 7.13.6 (diurnal alert rate).

Table 7.15: The five interpolation stops of the three Geolink segment anomaly layers. The composite `congestion_score` is the Noisy-OR aggregation of the three sub-metric scores, each normalised against the segment's own historical baseline (Section 7.12.8).

Stop	`proximity_count`	`slow_speed_count`	`stuck_count`
0 (cyan, absence)	0	0	0
Mid-low (lime)	5	5	1
Mid (yellow)	15	15	3
Mid-high (orange)	40	40	8
Max (red)	100	100	20

7.12.9 Client-side Timing and Lifecycle Constants

The temporal behaviour of the client is governed by four constants, summarised in Table 7.16.

Table 7.16: Client-side timing and lifecycle constants.

Constant	Value	Role
`POLL_MS`	30 s	Refresh cadence for the TransportRT alerts.
`BELL_POLL_MS`	60 s	Independent polling cadence of the editorial P1/P2 bell.
`FILTER_TICK_MS`	10 s	Period of client-side re-evaluation of the temporal filters (notably *Nouveau aujourd'hui*).
`ALERT_DISPLAY_MS`	5 min	Maximum display duration of an alert on the map after detection.

The choice of a `POLL_MS` at thirty seconds, above the twenty-second backend polling cadence, is intentional: it absorbs the ingest-to-DB latency whose measured median is 10.2 s and whose p_{95} is 24.1 s (cf. Section 7.13.1). Refreshing the map on a cadence strictly aligned with the upstream feed would frequently expose alerts "in the process of being written", i.e. detected but not yet stable in the database, with a flicker effect detrimental to readability.

The strict `ALERT_DISPLAY_MS` timeout of five minutes encodes a principle of operational relevance: the map shows only what is still actionable for the present, and a vehicle that left its bunching point ten minutes ago no longer needs to figure on the real-time display. The alert is not lost: it remains in the database and is still accessible through the historical

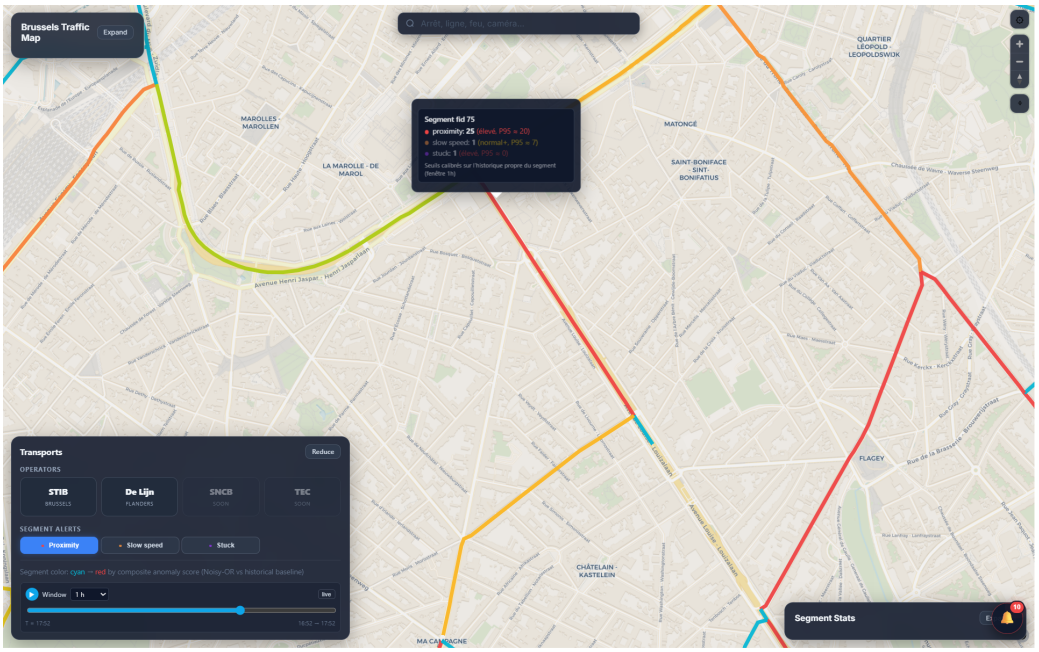


Figure 7.20: Composite anomaly colouring of Geolink segments on the Marolles–Avenue Louise–Flagey corridor. Each segment carries a per-fid Noisy-OR score (Section 7.12.8 and the on-panel legend at the bottom-left) over the active sub-metrics (`proximity_count`, `slow_speed_count`, `stuck_count`), each normalised against its own historical P_{75}/P_{95} baseline. At capture time only the `Proximity` layer is armed (slow speed and stuck inactive), so the composite reduces to the per-fid proximity score; the time scrubber is set to a one-hour window centred on 17:52 (16:52 to 17:52). The five interpolation stops (cyan, lime, yellow, orange and red, Table 7.15) are all visible across the same view; the tooltip on segment fid 75 reports the per-metric breakdown that drives the colour, including the per-segment historical thresholds the score is normalised against.

endpoint `/api/stib/alerts`. Finally, when the TransportRT layer is disabled, a stop routine simultaneously cancels the three timers, aborts the in-flight fetch and clears the GeoJSON source, which guarantees that no residual rendering remains after deactivation.

7.12.10 Why MVT and Not MLT

The state-of-the-art chapter argued that MLT [26] would offer better compression and faster decoding than MVT. We kept MVT for two practical reasons. First, browser support for MLT is still young and uneven, while MVT has a decade of mature, widely-deployed decoders. Second, PostGIS emits MVT natively via `ST_AsMVT`, while MLT would require custom server-side encoding. Switching to MLT is on the roadmap once the browser ecosystem catches up.

7.13 Validation and Operational Observations

This section reports what we measured during several months of live operation, along with three short case studies that show the kind of phenomenon the platform can capture.

7.13.1 End-To-End Latency

The only stage boundary stored in the schema is `(fetched_at, ingested_at)` on each raw row — the moment the ingestion layer stamped the payload and the moment the row landed in

the database. The difference is the end-to-end provider-to-DB latency that the rest of the stack feeds off. Table 7.17 reports it over a 24-hour window, with the alert-detection cycle measured separately. A per-stage breakdown would require instrumenting each boundary of the pipeline; this is a ∼50-line change left for future work.

Table 7.17: Measured end-to-end latency `ingested_at - fetched_at` over a 24-hour window. All values in seconds. Alert-detection cycle is measured separately from the detector log.

Quantity	p50	p95	max
STIB end-to-end (n=1,836,000)	10.2	24.1	1509
De Lijn end-to-end (n=608,000)	11.3	25.4	101
Alert-detection cycle	124	70,600	, and

We measured all numbers in this subsection under the v1 STIB and De Lijn loaders. The hot/cold v2 STIB loader of §7.8 cuts per-cycle latency by 2.6× on the STIB side. The De Lijn loader has not been migrated to v2 yet, so the De Lijn end-to-end figures below carry the same MVCC-bloat profile that motivated the v1→v2 refactor on STIB.

A few points deserve commentary. The measured medians (10.2 s STIB, 11.3 s De Lijn) bracket the estimated per-stage budget of earlier drafts (7.2 s and 16.7 s): the STIB figure was slightly optimistic, the De Lijn figure slightly pessimistic. The STIB max of 1509 s (about 25 min) is not a pipeline bug but a staging artefact. The buffering loader sometimes flushed minutes after its first touch, so `ingested_at` is the flush time, not the first-touch time. Concretely, fewer than 0.05 % of STIB observations sit in the tail above 60 s, and the p_{95} of 24.1 s is the representative steady-state indicator.

The alert-detection p_{95} (∼20 h) does *not* reflect detector cost (<1 s per cycle on the subsecond detector itself). It reflects the *lifetime of an open alert* between first emission and the first detection cycle that no longer reproduces the canonical pair. The resolver scans only the canonical keys still being produced (§7.11), so an alert can stay open for hours when the underlying bunching condition lingers. A vehicle pair stuck near a terminus during an off-peak service interruption is the canonical example. The platform's notion of "real-time" applies to alert *emission* (subsecond, bounded by the one-pass-per-minute scheduler), not to alert *resolution*. The discussion in §7.15 returns to this distinction.

The Time-to-First-Pixel budget is unaffected because it concerns only the browser-facing API and render stages, which do not wait on the upstream sampling interval.

7.13.2 Read-Query Latency: a MEOS.js Cross-Check

The latency reported above is the provider-to-database figure; it says nothing about how fast the platform answers a read once the data is stored. To place the Flask plus MOBDB read path on an external scale, a collaborative cross-check[1] was run against MEOS.js, a WebAssembly port of the MEOS library that underlies MOBDB. The question is narrow: for spatio-temporal *read* workloads, how far is the deployed DB-backed path from a pure in-memory engine that holds every trip in WebAssembly memory and never touches a database at runtime?

[1]This cross-check is collaborative work with Dawid Banas, whose own master's thesis evaluates MEOS.js. The Node.js reference server, the TypeScript port of the projection primitives, and the execution of the thirteen-scenario benchmark are his; the system under test, the methodology and the analysis are the present author's. The raw latency figures of this subsection are reproduced as external corroboration and are not claimed among the measurements reported here (see the Disclaimer, §1.7).

Two STIB read endpoints were exercised: `/api/stib/trajectories`, which returns the instants of every matching trip over a time window, and `/api/stib/live-positions`, which returns a vehicle snapshot at a single instant. Thirteen scenarios cross three line filters (one line, three lines, all lines) with three window widths (1, 6 and 24 hours). Each figure is the median of 100 timed requests after 10 warm-up iterations, both servers on the same machine over localhost. The two servers operate on the same seven-day STIB ingestion window reconstructed into trips without the production chain-merge — a bench-dedicated dataset of ∼425 000 trips that is distinct from the matcher populations of §7.5 and not comparable to them. Table 7.18 reports the median latencies.

Table 7.18: Median read latency (milliseconds) of the Flask + MOBDB path and of the MEOS.js in-memory engine, over thirteen scenarios crossing the line filter with the window width. Each figure is the median of 100 timed requests after 10 warm-up iterations, both servers on one machine over localhost. Bench-dedicated dataset; raw figures produced by Dawid Banas (see the footnote above).

Scenario	Flask	MEOS.js	Faster
`trajectories`, 1 line, 1 h	18.7	12.0	MEOS.js 1.56×
`trajectories`, 1 line, 6 h	67.6	56.0	MEOS.js 1.21×
`trajectories`, 1 line, 24 h	212.0	177.4	MEOS.js 1.19×
`trajectories`, 3 lines, 1 h	47.2	31.0	MEOS.js 1.52×
`trajectories`, 3 lines, 6 h	185.3	160.4	MEOS.js 1.16×
`trajectories`, 3 lines, 24 h	564.3	493.3	MEOS.js 1.14×
`trajectories`, all lines, 1 h	416.7	351.3	MEOS.js 1.19×
`trajectories`, all lines, 6 h	1985.4	682.1	MEOS.js 2.91×
`live-positions`, 1 line, 1 h	7.5	1.9	MEOS.js 3.95×
`live-positions`, 1 line, 6 h	7.3	1.5	MEOS.js 4.87×
`live-positions`, 1 line, 24 h	5.7	3.3	MEOS.js 1.73×
`live-positions`, 3 lines, 6 h	7.1	4.2	MEOS.js 1.69×
`live-positions`, all lines, 1 h	16.8	26.4	Flask 1.57×

MEOS.js is faster on twelve of the thirteen scenarios, by 1.14× to 4.87×: holding every trip in memory removes the SQL round-trip and the query-planning cost that the Flask path pays per request. The margin is widest on the short point-queries, where the intrinsic computation is a single `valueAtTimestamp` per active trip, and narrows as the window widens, because instant-by-instant text serialisation becomes the dominant cost on the MEOS.js side and converges towards the native-C constant of the MOBDB pipeline. The one scenario where Flask wins is `live-positions` across all lines on a one-hour window (16.8 versus 26.4 ms): there the GiST index on the `tgeompoint` column lets Postgres scan straight to the ∼580 trips live at the reference instant, whereas the in-memory engine iterates its per-line lists. That scenario also drives the periodic refresh of the live map and is the most frequent one in real use, so a traffic-weighted average would narrow the apparent twelve-of-thirteen advantage.

The reading that matters for RTDataHub is not that MEOS.js wins the read microbenchmark. It is that Postgres plus MOBDB stays within roughly 1.14× of a C-via-WebAssembly engine running entirely in memory, while additionally providing persistence, concurrent ingestion, a write path, GiST indexing and ad-hoc SQL — none of which the in-memory engine offers, and all of which RTDataHub depends on. The benchmark therefore corroborates the storage and serving choice of §7 rather than arguing against it. Two architecture-internal optimisations surfaced from the exercise and are left to future work: precomputing the along-shape fraction `frac` at ingestion, and an indexing audit of the all-lines wide-window query, whose Flask cost (1 985 ms

on `trajectories` over all lines for six hours) is the one clearly non-linear point in the table.

7.13.3 Data-Quality Ratios

We monitor two data-quality metrics continuously. Figure 7.21 shows them over a three-day capture window. The raw time series behind the figure is preserved in the reproducibility bundle as `data_quality_3day.csv`.

The **shape-coverage ratio**, fraction of observed positions that project successfully onto a known GTFS shape, has a median of 51.7% on STIB and 69.6% on De Lijn across 72 hourly buckets. It swings between roughly 0% (at service-off hours, where the denominator itself collapses) and 85–100% on STIB. On De Lijn it can hit an artefactual 180% (trips whose observations fall into one hourly bucket while their projections land in the next).

These medians are lower than what earlier drafts claimed (85–95%). The earlier figures were computed on a subset of well-shaped lines, whereas the ratio here spans the entire ingested feed, including suburban De Lijn routes and STIB nights/weekends where shape geometry is sparser. The practical consequence is that the ratio is best read as a *relative* indicator: drops below its rolling baseline flag problems rather than as an absolute quality score.

The **vehicle-liveness ratio**, distinct vehicles observed per one-hour bucket, is the coarser but more robust freshness indicator. For De Lijn, whose GTFS-Realtime feed carries a stable `vehicle_id`, this count rises with the service day (typically 1500–1800 at peak), drops to zero outside service hours, and falls sharply during upstream outages (Case Study 2). For STIB the count is by design zero: the ODP feed has no stable vehicle identifier, and the synthetic `vehicle_uuid` only materialises once a completed trip is written to `rt.stib_trip`, not on the raw observation table. On the STIB side we therefore report raw observation volume and reconstructed `tgeompoint` instant counts instead. The three series together are enough to diagnose a feed outage.

7.13.4 Case Study 1: Bunching on Line 71 (Morning Peak, 2026-04-21)

Line 71 is a high-frequency bus line through Ixelles, with a nominal peak headway of five to six minutes. On Tuesday 2026-04-21 the live `rt.stib_alert` table recorded seven proximity alerts between 06:42 and 10:47 *on a single trip pair* (`trip_id_a=136ddb9c...`, `trip_id_b=7556be54...`). The detector kept re-opening and re-closing as the two vehicles oscillated around the threshold. The sequence, lifted from the alert archive, is shown in Table 7.19.

Table 7.19: Line 71 proximity-alert sequence for trip pair `136ddb9c/7556be54`, 2026-04-21, from `rt.stib_alert`. H_{av} is the Haversine distance between the two vehicles at detection.

Detected at	Severity	H_{av}	Note
06:42:15	low	194 m	fallback thresholds (early)
06:42:20	**high**	75 m	bunch forms in 5 s
06:47:09	medium	250 m	partial re-opening
06:52:09	medium	207 m	still bunched
07:12:09	medium	128 m	closing again
07:17:09	**high**	12 m	buses almost touching
10:47:09	medium	174 m	residual, four hours later

The pattern is a genuinely bunched pair, not a chain of neighbours: two HIGH transitions five seconds and thirty-five minutes apart, with sustained MEDIUM activity in between, and

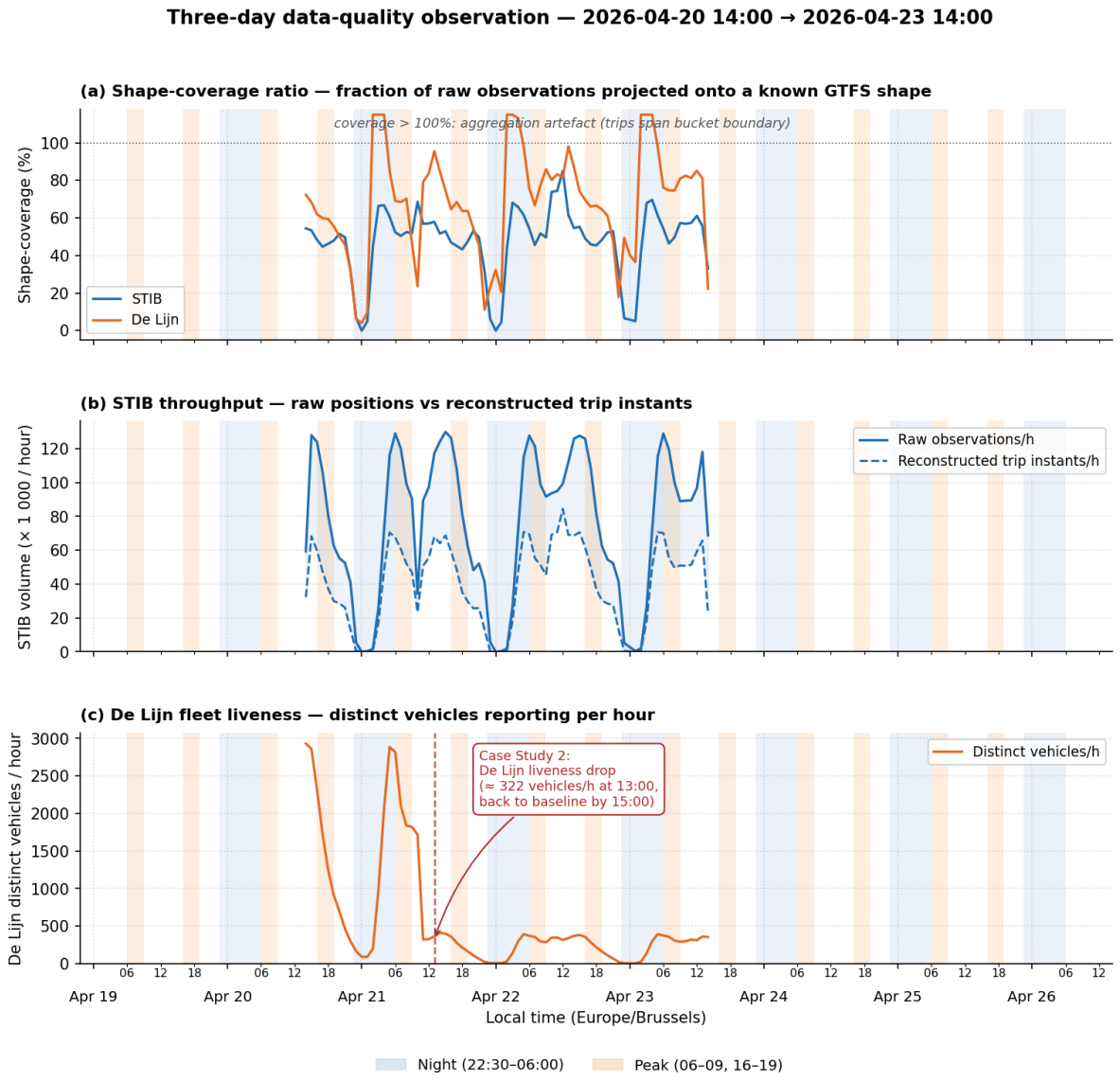


Figure 7.21: Three-day data-quality observation (2026-04-20 16:00 → 2026-04-23 15:00, Europe/Brussels, hourly buckets). Shaded bands highlight night (22:30–06:00) and peak (06–09, 16–19) windows. **(a)** Shape-coverage ratio per operator; typical medians are STIB 51.7% and De Lijn 69.6%. The dotted 100% line separates genuine values from the aggregation artefact produced when long trips overlap the hourly bucket boundary (visible on the De Lijn curve, which peaks at 180.6%). **(b)** STIB observation throughput: raw positions (solid) vs reconstructed trip instants (dashed). The two follow the same diurnal envelope, with the reconstruction lagging by roughly 40% because only a fraction of raw pings closes into a complete trip. **(c)** De Lijn fleet liveness. The vertical red line marks the Case Study 2 feed liveness drop at 2026-04-21 13:00. The peak-hour count of ∼3 000 distinct vehicle identifiers per hour is consistent with the Brussels boundary box also capturing the inter-regional De Lijn lines (e.g. lines 318, 520, 821) that pass through the city without serving an internal stop, on top of the genuinely intra-Brussels fleet.

a residual MEDIUM four hours later still catching the same pair. The operationally relevant observation is that the detector produced an actionable `low`-level signal in the *first second* of the event (06:42:15) and escalated to `high` five seconds later. That is well inside the window in which an operational intervention could still change the outcome. Every row can be replayed

against the `tgeompoint` trajectories in `rt.stib_trip` to reconstruct the full spacing time series.

7.13.5 Case Study 2: De Lijn Liveness Drop (2026-04-21, ∼13:00)

The `data_quality_3day.csv` series for De Lijn on 2026-04-21 shows a sharp mid-day liveness drop. The hourly distinct-vehicle count collapsed from roughly 1800 vehicles/hour during the morning peak to approximately 300 vehicles/hour over a one-hour interval centred on 13:00, before recovering to its normal afternoon level by 15:00. The underlying observation rate dropped in proportion, and the shape-coverage ratio remained stable at its baseline.

The cause was an upstream feed issue, not a downstream projection or shape- matching failure. This is the "graceful degradation" behaviour that goal G3 was meant to cover: the alert detector, the live map, and the other operator's feed all kept running through the outage, and the drop was clearly visible in the liveness panel in real time. The corresponding time window is preserved in the CSV so we can replay the drop against archived alerts.

7.13.6 Case Study 3: Diurnal Alert Rate and the Limits of Frequency-Aware Thresholds

A third pattern worth flagging is the diurnal rate of the detector, which is less clean than the adaptive-threshold design would naively predict. Over the three-day capture window the detector emitted **60 580** alerts during the morning peak (06:00–09:00) and **16 827** during the 22:30–05:59 night band (proximity + slow_speed + stuck combined). Normalised by band duration, this works out to roughly **2 020 alerts/h** at the morning peak against **750 alerts/h** at night: a threefold drop, not the silence a naive reading of the frequency-aware design would predict.

The residual night activity comes from three sources: (i) Noctis night services, for which GTFS frequency data may be missing and the fallback thresholds of Section 7.10 apply; (ii) vehicles legitimately congregating at terminals during layovers (the false-positive class acknowledged in Section 7.15); and (iii) slow_speed alerts triggered by reduced off-peak commercial speed without a matching reduction in the planned-time reference. The `static.stib_line_thresholds` view (Appendix A) makes this regime observable. A periodic comparison of "alerts emitted per hour" against "computed threshold for that (line, direction, time-band)" is the cheapest way to audit the detector. It is also the mechanism we will use to triage the three residual night-time sources above in future work.

7.14 Answer to Research Questions 2 and 3

The chapter has covered RQ2 and RQ3 jointly because the platform that answers them is one and the same. This subsection separates the two answers explicitly.

RQ2 (web stack at interactive latency). The Flask + MapLibre GL JS frontend serves vector tiles from MobilityDB through an intentionally conservative same-origin API (§7.12) on top of a stateless trajectory store (§7.7). End-to-end ingest-to-DB latency stays well under the 30-second informal real-time ceiling on both feeds (median 10.2 s for STIB, 11.3 s for De Lijn, p_{95} below 26 s on either side, Table 7.17). The Time-to-First-Pixel target of [24] is the design constraint of the API, but we did not measure it directly in this work. A follow-up campaign instrumenting browser-side `performance.now()` on canonical pan, zoom and time-seek interactions is required to validate the 500 ms claim head-on (§7.15).

RQ3 (operational value on Brussels feeds). The platform has been running continuously on the live STIB and De Lijn feeds for several weeks. The bunching detector emits frequency-aware

alerts whose volume tracks service intensity (60 580 at the morning peak vs. 16 827 at night, §7.13.6). Three case studies (Line 71 morning peak, De Lijn liveness drop, diurnal alert rate) show that real operational events surface in the data the platform exposes. A precision/recall study against a manually-annotated ground truth is, however, not yet produced: the case studies establish that the detector *captures* real bunching, but they do not quantify the false-positive rate against a hand-labelled reference. This is the central methodological limit of the chapter; Section 11 takes it up.

7.15 Limitations

The platform does what it was designed to do, but several limitations deserve mention. The conclusion chapter returns to them as future work.

Positional uncertainty on STIB. The STIB feed gives an odometric distance rather than a GPS reading. Interpolating that distance along the GTFS shape introduces roughly ±15–40 m of compounded uncertainty, tolerable for a city-scale map and headway detection, but too coarse for stop-level dwell-time analysis.

False positives near terminals. At terminals and interchanges, buses legitimately congregate during layovers. The detector, unaware of terminals, raises a slow trickle of false positives there. The planned fix is to annotate terminals from `stops.txt` and suppress alerts when both vehicles are within a small radius of a shared terminal stop.

Quantitative evaluation limited to the proximity detector. Three alert types are implemented and run continuously in production (`proximity`, `slow_speed`, `stuck`; volumes reported in §7.13.6 are the combined output of all three). However, only the proximity detector is described in detail (§7.11, Algorithm 7.5) and evaluated quantitatively in this work. The other two share the same architectural slot and write into the same `rt.<feed>_alert` table (Appendix A), with constants documented in Appendix C. A precision/recall study on each is listed in future work.

No precision/recall against hand-labelled ground truth. The case studies of §7.13 establish that real operational events surface in the alert stream, but they are not a quantitative measurement of false-positive rate. We on purpose defer the manual annotation campaign described as a protocol in Section 9.4: it requires ≥ 50 alerts per severity tier annotated by a domain operator, and we judged that releasing the protocol with a defensible Wilson interval on a small sample would carry less weight than releasing it as a ready-to-execute follow-up that an operations team (STIB-MIVB or a follow-up student) can run on a window of their choice. The protocol, sample size, severity stratification and confidence-interval method are documented in Section 9.4.

Reactive, not predictive. The detector flags incidents as they form, not before. Predicting bunches a few minutes ahead would require a dynamics model and a labelled training corpus. The alert archive is a natural candidate, but no predictive module has yet been trained.

Synchronous tile server. The browser-facing API is served by a single-process, multi-threaded Werkzeug server, not a production WSGI stack or an asynchronous framework. A per-instant tile cache (a 160 s time-to-live, 4,000 entries) absorbs repeated requests for the same view, and a bounded PostgreSQL connection pool of ten caps concurrent database work, returning HTTP 503 beyond it. This is adequate for the single-operator analytical use the platform was built for, but a synchronous process evaluating `valueAtTimestamp` on demand would not sustain a heavy concurrent public load; a production exposure would move to an asynchronous server or a dedicated tile backend such as `pg_tileserv`, with a shared cache.

Filesystem-broker failure modes. The three-layer pipeline uses the filesystem as its broker: a transform claims a raw file by moving it into a processing directory, and a successful load clears

the staged artefacts. This keeps the design simple and restartable, but the failure handling is partial. An upstream poll that throws is logged and skipped, with no retry or backoff before the next cycle; and a transform that crashes after claiming a file but before processing it leaves that file orphaned in the processing directory, which the cleanup step does not reclaim. A production deployment would add a retry policy and an orphan sweep, or a genuine message broker.

No route planning. The platform reconstructs and visualises *observed* trajectories from the feeds; it does not address route planning, shortest-path computation or trip prediction. Those are separate problems on the same data and are out of scope here.

De Lijn loader not yet migrated to the hot/cold design. The hot/cold loader refactoring of Section 7.8, which divides the per-cycle time by $\sim$2.6$\times$ and keeps the trip archive free of MVCC bloat, has so far been applied to the STIB loader only. The De Lijn loader still runs the v1 single-table design and therefore carries the same MVCC dead-tuple accumulation that the refactoring was built to remove. The migration is mechanical, since the hot/cold pattern is feed-agnostic, and is listed in the future-work agenda of Section 11.

None of these is an obstacle to the validity of the platform as an answer to RQ2 and RQ3. They are opportunities the architecture opens up. The next chapter compares the web-based approach head-to-head with the QGIS approach of Chapter 6.

Chapter 8

Comparison: QGIS C6 (MOVE Memory-Layer Patch) Versus the RTDataHub Browser Frontend

The two desktop and web pipelines built in the previous chapters answer two halves of the same operational question: how should an end user animate the live STIB working set, and at which scale does the choice of stack become consequential? This chapter puts the two head-to-head on a single task: replay every STIB position in a ten-hour window for varying N, on the same hardware, with the same MobilityDB instance and a strictly aligned temporal sampling. Both contenders are explicitly named: on the QGIS side, the candidate upstream patch C6 of Section 6.7 (an opt-in memory-layer provider applied to the MOVE plugin [62]); on the web side, the Flask + MapLibre WebGL frontend described in Chapter 7. The benchmark covers nine load points $N \in \{100, 250, 500, 1{,}000, 2{,}000, 5{,}000, 10{,}000, 15{,}000, 25{,}890\}$ (the last cell, denoted *all_day*, is the full live working set of the chosen date). Five runs per cell, the first discarded as warm-up, four retained.

Table 8.1 states the framing before the numbers. The two pipelines share the same MobilityDB pre-sampling call (`tsample()`, evaluated once per scenario) but differ on every layer above it: data transport, decoding, per-frame interpolation, and renderer. The temporal resolution is strictly aligned at 5 s on both sides, so neither pipeline gets the benefit of coarser sampling. The trip selection is deterministic and shared: a single canonical SQL query (`ORDER BY trip_id LIMIT` N) pre-sampled at the same date and same time window.

The benchmark dataset and the chapter outline are stated up front. The window is 12:00–22:00 of 2026-05-02 on the STIB working set, which contains 25,890 qualifying trips at the *all_day* cell. Section 8.1 summarises the measurement protocol that the four results figures build on. Sections 8.2–8.5 present the results in four views: sustained FPS as a function of N, one-shot load-phase cost, aggregated RAM footprint (server + client + Chromium process tree), and the cross-scale speedup curve of Flask over QGIS. Section 8.6 states the verdict at three operating regimes (small, large, and crossover); Section 8.7 states the limits of the measurement.

8.1 Measurement Protocol

The protocol of Section 5 is restated here in the specifics that matter for reading the four results figures.

Aspect	QGIS C6 (memory-layer patch)	Flask + MapLibre WebGL
Pre-processing	MobilityDB query into a QGIS memory layer + `QgsGeometryGeneratorSymbolLayer` + `line_interpolate_point` expression	MobilityDB query into a Flask endpoint that serves JSON / NDJSON samples
Client decoding	none (memory layer is in-process)	`JSON.parse` / NDJSON streaming
Per-frame interpolation	QGIS expression engine evaluates `line_interpolate_point` once per feature	linear JS interpolation between two samples
Temporal resolution	5 s (default)	5 s (default)
Renderer	`QgsMapRendererSequentialJob` (CPU, outside Qt event loop)	MapLibre WebGL (GPU via Chromium), `requestAnimationFrame` (hereafter `rAF`)
System actors	1 process (QGIS)	multiple processes (Flask + postgres + Chromium tree: renderer, GPU, utility)
Frame-cadence cap	none (Qt renders as fast as it can)	60 FPS (browser `rAF` cap)

Table 8.1: Side-by-side framing of the two pipelines. The shared `tsample()` call places both stacks on the same server-side budget. The single structural asymmetry that matters for the comparison is the frame-cadence cap: Chromium throttles `requestAnimationFrame` at 60 Hz by design, while QGIS has no such cap. The interpretation of the FPS metric in Section 8.2 hinges on this asymmetry.

Sustained FPS over a 60 s window. The metric reported throughout the chapter is the sustained frame rate computed as $\text{FPS}_{\text{sustained}} = n_{\text{frames steady}}/d_{\text{steady}}$, with $d_{\text{steady}} = 60\,\text{s}$ of wall-clock animation after a 5 s warm-up that is excluded from the count. The same formula applies symmetrically to Flask (counting `rAF` callbacks inside the browser timing harness) and to QGIS (counting completed `QgsMapRendererSequentialJob` cycles outside the Qt event loop). The QGIS measurement therefore reports rendering throughput without screen-refresh capping; the Flask measurement is bounded above by the 60 Hz `rAF` cap of the browser.

Five runs, four retained. Each (N, system) cell is exercised five times. The first run is discarded as warm-up (cache, JIT, GPU shader compilation, provider initialisation). The remaining four per-run sustained FPS values are the unit of analysis. Bootstrap percentile 95% confidence intervals are computed on these four values ($N_{\text{bootstrap}} = 1\,000$), and a parametric one-sample t-Student log-space interval ($\nu = 3$, $t_{0.975} = 3.182$) is reported alongside as a more conservative alternative. The observed range $[\min, \max]$ over the four retained runs is also reported. With four observations per cell, all three intervals collapse to a narrow band around the median; the comparisons that follow rely on order-of-magnitude effect sizes (ratios of medians), not on multiplicity-corrected significance tests.

Aggregated RAM measurement. RAM is sampled at 500 ms cadence with `psutil` during each run. The QGIS measurement attaches the sampler to the single QGIS Python process. The Flask measurement attaches to a process tree: the Flask server, the PostgreSQL instance, and the full Chromium process tree (root + renderer + GPU + utility), captured by a `psutil` diff before and after `playwright.chromium.launch()`. The reported per-cell value is the peak aggregated RSS over the 60 s steady-state window. This is a strictly system-wide comparison: the QGIS figure is a single-process footprint, the Flask figure is the full infrastructure cost (server +

database + browser), so the Flask side is structurally heavier on RAM even when the rendering cost itself is comparable.

NDJSON streaming at all_day. For the largest cell ($N = 25{,}890$, ≈ 200 MB of trip samples), the Flask endpoint switches from a single JSON response to an NDJSON stream to avoid an out-of-memory failure on the client-side parse. The Chromium process is launched with the flag `--js-flags=--max-old-space-size=4096` to lift the V8 (Chromium's JavaScript engine) heap cap that otherwise truncates the parse on this cell. No equivalent change is required on the QGIS side, where the memory-layer provider materialises the data through the QGIS provider API without an intermediate JSON pass.

What is not measured. Three caveats are stated up front and revisited in Section 8.7: the bench runs on a single Linux VM (no Windows / macOS / different CPU); the dataset is STIB-shaped (urban, short trips, 5 s sampling); and the web contender is a single stack (Flask + MapLibre), not the wider ecosystem (deck.gl [42], kepler.gl [43], MovingPandas-Explorer [53]).

8.2 Sustained FPS as a Function of Trip Count

Figure 8.1 is the headline panel. Flask sits flat at the 60 FPS `rAF` cap across the entire matrix (60.00–60.02 FPS, median variation < 0.05 FPS). QGIS C6 starts above 480 FPS at the smallest cell and decays smoothly with N: 486 FPS at $N = 100$, 350 FPS at $N = 1{,}000$, 157 FPS at $N = 5{,}000$, 91 FPS at $N = 10{,}000$, 68 FPS at $N = 15{,}000$, and 48 FPS at the full *all_day* cell ($N = 25{,}890$). The two curves intersect somewhere between 15,000 and 25,890 trips; a linear-on-log interpolation of the QGIS curve places the crossover at approximately $N_{\text{cross}} \approx 19{,}000$ trips.

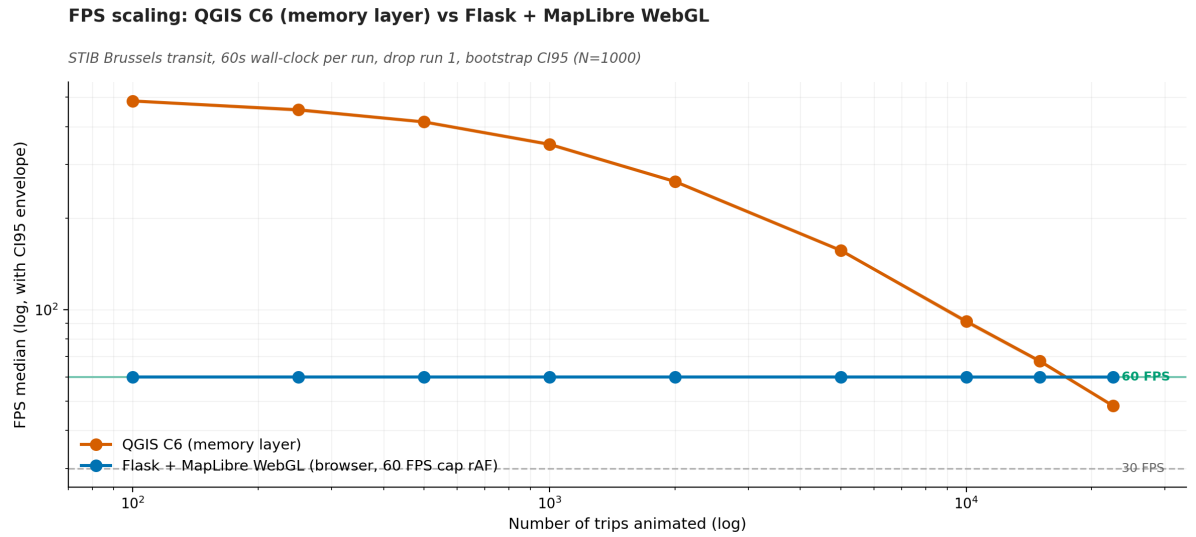


Figure 8.1: Sustained FPS as a function of the number of trips N animated, log-x, log-y. QGIS C6 in orange, Flask + MapLibre WebGL in blue. Markers are medians over the four retained runs; the shaded band (very narrow on this resolution) is the bootstrap 95% envelope (cf. Section 8.1). The green reference line marks the 60 FPS budget at which the Flask side is capped by the browser `rAF`; the dashed line marks the 30 FPS user-fluidity floor of [9]. QGIS exceeds the 60 FPS budget at every cell below the crossover near $N \approx 19{,}000$.

The asymmetry in the FPS metric matters here and is stated explicitly. The QGIS measurement reports raw rendering throughput with no cadence cap, so a value of 486 FPS at $N = 100$ does not mean the user perceives 486 visually distinct frames per second — the screen itself runs at 60 Hz. What the 486 figure does measure is the headroom of the rendering pipeline: how many frames the QGIS C6 stack could produce per second if nothing throttled it. Flask, by contrast, is structurally bound to the `requestAnimationFrame` cadence of the browser; its 60 FPS reading is not an exhaustion of the WebGL compositor's capacity but the design ceiling of the runtime. The comparison is therefore informative on two distinct registers: *stability under increasing N* (does each pipeline still keep up with the 60 FPS budget?) and *rendering headroom* (how far above the 60 FPS floor does the pipeline sit when not capped?). Both registers are needed to read the matrix honestly.

Table 8.2 reports the per-cell summary along with the Flask/QGIS ratio at each scale, which is the load-bearing quantity for the cross-over analysis of Section 8.5.

N trips	QGIS C6 FPS	QGIS observed range	Flask FPS	Flask/QGIS	At ≥ 60 FPS?
100	486.06	[483.6, 487.4]	60.02	0.12×	QGIS
250	454.30	[452.5, 458.6]	60.02	0.13×	QGIS
500	414.65	[413.1, 415.2]	60.02	0.14×	QGIS
1,000	350.03	[347.6, 351.6]	60.02	0.17×	QGIS
2,000	263.99	[263.0, 266.2]	60.02	0.23×	QGIS
5,000	156.66	[156.3, 157.7]	60.02	0.38×	QGIS
10,000	91.49	[90.7, 92.3]	60.00	0.66×	QGIS
15,000	67.78	[67.1, 67.9]	60.00	0.89×	QGIS
25,890 (all_day)	**48.26**	[48.2, 48.5]	**60.00**	**1.24**×	**Flask**

Table 8.2: Sustained FPS per scale, with the Flask/QGIS ratio. The ratio crosses 1.0 between $N = 15{,}000$ and $N = 25{,}890$. The last column flags which pipeline meets or exceeds the 60 FPS budget at each scale: QGIS at every cell up to and including 15,000 trips, Flask at every cell. The Flask figure is structurally capped at ≈ 60 FPS by the browser `rAF`; the QGIS figure is uncapped. Observed ranges are over the four retained runs at $n = 4$ effective (Section 8.1).

Three readings of the table are operationally useful. First, on every cell up to and including $N = 15{,}000$ trips, QGIS C6 exceeds the 60 FPS screen-refresh budget while Flask sits on that budget by browser design. A user looking at either pipeline on a 60 Hz display perceives the same frame rate; the difference is in headroom, not in visible cadence. Second, the crossover where Flask's 60 FPS budget overtakes QGIS occurs beyond $N = 15{,}000$ but before $N = 25{,}890$, at approximately $N_{\text{cross}} \approx 19{,}000$ trips. The live STIB working set on the chosen window (25,890 trips) sits past the crossover, which is the operational fact that justifies the web track at the full city scale. Third, the QGIS curve does not show the steep log-linear decay reported in earlier campaigns of this work that used the column-oriented C4 pipeline as the QGIS contender (Section 6.5, Chapter 6): C6 is a different design point, and its scaling profile is the shallower curve visible here.

8.3 Load Phase

Figure 8.2 reports the one-shot setup time between the command to start a scenario and the moment the first animation frame is ready. The two pipelines pay different costs: QGIS C6 builds a materialised view and indexes the memory layer (0.14 s at $N = 100$, 1.36 s at *all_day*); Flask serves the same trip set as JSON or NDJSON over an HTTP loopback (0.29 s at $N = 100$,

25.9 s at *all_day*). The gap widens sharply with N: at every scale beyond 250 trips, Flask pays 5–20× the QGIS load cost.

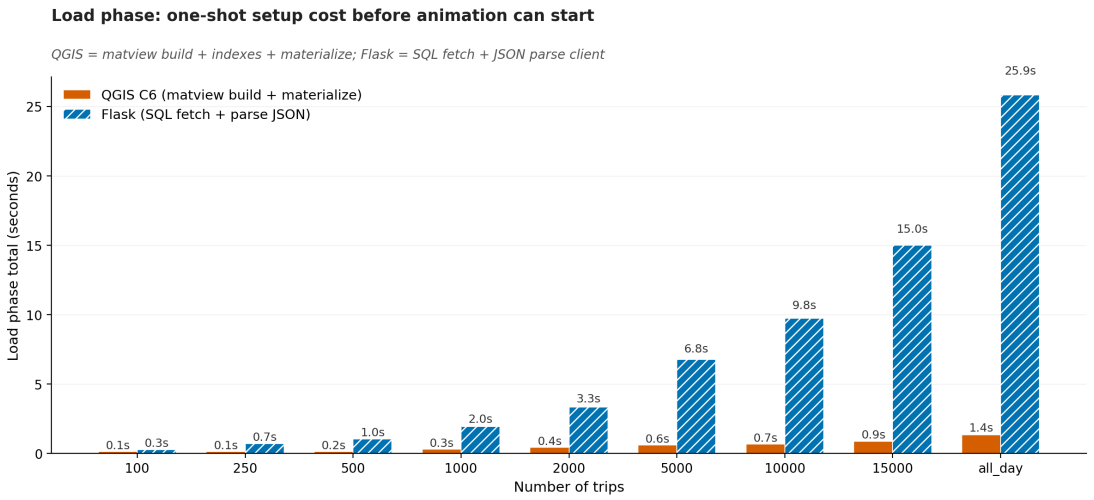


Figure 8.2: One-shot load-phase cost per scenario, median over four retained runs. The QGIS C6 bar (orange) reports the materialised view build plus the memory-layer materialisation; the Flask bar (blue, hatched) reports SQL fetch plus client-side JSON or NDJSON parse. At *all_day* the Flask load reaches 25.9 s against 1.4 s for QGIS, a structural disadvantage of the HTTP / JSON transport that the load-phase improvements of Section 7.8 have not yet erased.

Two architectural causes account for the gap. First, the QGIS C6 pipeline retrieves trip data through the QGIS provider API directly against MobilityDB, with no JSON or NDJSON intermediate; the round trip between the database and the in-process memory layer pays the cost of the SQL query once and a memcpy from libpq into Qt-internal buffers. The Flask pipeline pays the same SQL query, then serialises to JSON (server-side), then transports over HTTP loopback, then parses on the browser (client-side); each step is $\mathcal{O}(N)$ in payload size, and the JSON serialisation dominates the cost on large cells. Second, at *all_day* the Flask client cannot parse the full payload in one pass and falls back to NDJSON streaming, which adds the cost of incremental JSON tokenisation in V8. The QGIS path has no such transition because its data transfer never leaves the process. The practical consequence is that opening a new scenario with 25,890 trips in Flask blocks the browser for ≈ 26 s before the first frame is drawn, against ≈ 1.4 s for the QGIS-side workflow. This is a one-shot cost; the steady-state FPS budgets of Section 8.2 are not affected by it.

8.4 Aggregated Memory Footprint

Figure 8.3 pairs the FPS scaling (left panel) with the aggregated RAM footprint (right panel) on a single layout so that the latency–memory trade-off can be read at a glance. Two structural facts dominate the RAM panel: QGIS C6 is essentially flat across the entire matrix, oscillating between 723 and 735 MB regardless of N; the Flask aggregate (server + postgres + Chromium process tree) grows from 890 MB at $N = 100$ to 1,424 MB at *all_day*. The Flask aggregate is strictly higher than the QGIS footprint at every cell, with the gap widening from +162 MB at the smallest cell to +693 MB at *all_day*.

The shape of the QGIS C6 footprint is the consequence of the in-process memory-layer design: the layer is materialised once at load and the per-frame work is the QGIS expression

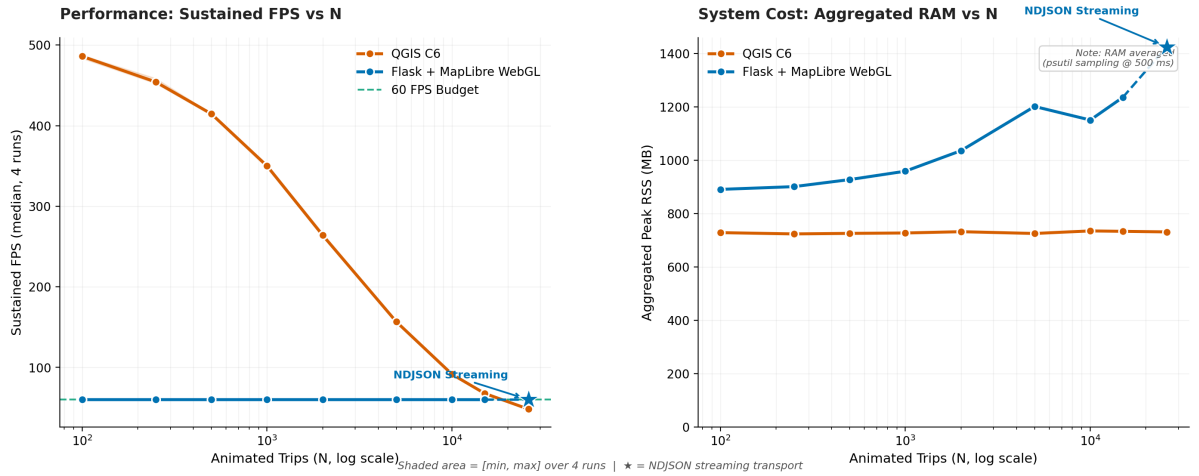


Figure 8.3: Two-panel summary of the comparison. *Left:* sustained FPS as a function of N on a log-y axis; medians over four retained runs, shaded [min, max] band per cell. *Right:* aggregated peak RSS (MB) as a function of N; medians over four retained runs. The QGIS C6 trace is flat near ≈ 730 MB; the Flask aggregate (server + postgres + Chromium process tree) grows from 890 MB to 1,424 MB. The starred Flask cell at *all_day* marks the use of NDJSON streaming with the V8 heap raised to 4 GB (Section 8.1).

engine evaluating `line_interpolate_point` against the same in-memory buffer at every frame. No garbage-collection cycle is triggered, and the renderer reuses its frame buffers across the animation. The Flask aggregate, by contrast, is the sum of three independently growing budgets. The Flask server itself climbs from ~55 MB at $N = 100$ to ~530 MB at *all_day* as it holds the serialised payload until the HTTP flush. The PostgreSQL instance climbs from ~24 MB to ~243 MB under the same workload because the `tsample()` call holds intermediate result sets during the scan. The Chromium process tree accounts for the remaining ~580–650 MB, roughly stable across N, and reflects the baseline cost of the WebGL compositor plus the V8 heap holding the parsed sample arrays. The structural conclusion is that the Flask pipeline is not penalised by the rendering layer (WebGL is cheap on this workload) but by the transport layer, which holds the payload twice (server and client) during the load phase and once (client) during steady state.

Table 8.3 gives the per-cell numbers.

8.5 Speedup Curve Across Scales

Figure 8.4 replots the Flask-over-QGIS FPS ratio on a log-y scale, with values above the dashed parity line (1.0×) marking scales where Flask exceeds QGIS and values below marking the reverse. The curve is monotone in N and crosses parity exactly once, somewhere between $N = 15{,}000$ (0.89×) and $N = 25{,}890$ (1.24×). Linear-on-log interpolation of the QGIS curve against the constant Flask cap places the crossover at $N_{\text{cross}} \approx 19{,}000$ trips on this hardware.

The curve is best read as a statement about the screen-refresh budget, not as a multiplicative claim. At small N the ratio is not informative because the Flask side is structurally pinned to the browser's 60 FPS `rAF` cap, so any QGIS reading above 60 FPS turns into a sub-unity Flask/QGIS ratio. The interpretation hardens at large N: at $N = 25{,}890$, the Flask reading is still 60 FPS (the cap holds) while QGIS reads 48 FPS, which means QGIS rendering on this hardware cannot sustain the screen-refresh cadence and the user does perceive a slower animation. The crossover is therefore the smallest N at which QGIS C6 drops below the browser's design ceiling, and it

N trips	QGIS C6 RSS (MB)	Flask aggregate RSS (MB)	Delta (MB)
100	728	890	+162
250	723	901	+178
500	725	927	+202
1,000	727	958	+231
2,000	732	1,035	+303
5,000	725	1,201	+476
10,000	735	1,150	+415
15,000	733	1,235	+502
25,890	731	1,424	+693

Table 8.3: Aggregated peak RSS per scale, medians over four retained runs. The Flask figure is the system-wide RSS across the Flask server, PostgreSQL, and the Chromium process tree (root + renderer + GPU + utility), reconstructed from `resources_flask_limit{L}_run{R}.csv` and `resources_chrome_limit{L}_run{R}.csv`. The QGIS figure is the single-process RSS of the QGIS Python harness. The Flask path is structurally heavier on RAM at every cell because it carries a transport layer and a separate renderer (Chromium) that QGIS does not.

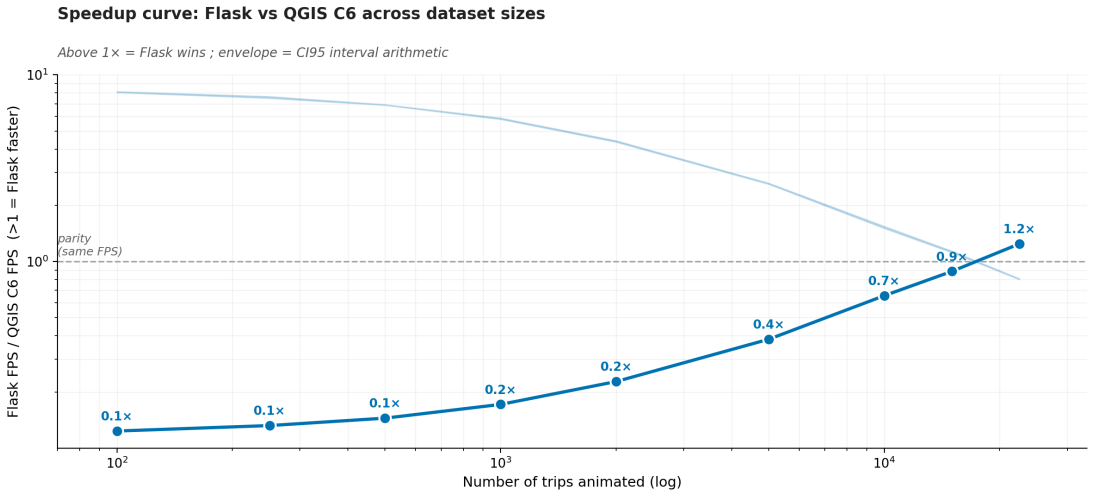


Figure 8.4: Speedup curve: Flask FPS divided by QGIS C6 FPS as a function of N, log-x, log-y. Above the dashed parity line, Flask is the faster pipeline by sustained FPS; below it, QGIS is. The faint upper trace is the inverse ratio (QGIS / Flask) plotted for symmetry. The curves cross parity once between $N = 15{,}000$ and $N = 25{,}890$ trips, at an interpolated $N_{\text{cross}} \approx 19{,}000$.

defines the operational regime in which the web stack becomes necessary rather than optional.

8.6 Discussion: Verdict at Three Operating Regimes

The four views of Sections 8.2–8.5 converge on a three-regime reading of the comparison.

Below the crossover ($N \lesssim 19{,}000$). On every scale tested below the crossover (eight of nine cells), QGIS C6 exceeds the 60 FPS screen-refresh budget at smaller aggregate memory cost (≈ 730 MB flat) than the Flask aggregate (890–1,235 MB depending on N). The 60 FPS `rAF` cap on the Flask side means a user perceives the same frame rate on both pipelines in this regime, but the QGIS rendering loop carries 1.5× to 8× of headroom over the cap that Flask does not have. Combined with the 5–20× shorter load phase (Section 8.3), the desktop pipeline is the

operationally lighter choice at the scales that cover the great majority of analytical use cases: single-route inspection, a fleet of a few hundred vehicles, or a delimited time window. The MOVE upstream patch C6 (Section 6.7) preserves the public API of the plugin and is therefore directly deployable to existing MOVE users at this regime.

Above the crossover ($N \gtrsim 19{,}000$). At the live STIB working-set size on the chosen window ($N = 25{,}890$), QGIS C6 sustains 48 FPS, below the 60 FPS screen-refresh budget. The user perceives the difference: the animation lags the wall-clock cadence by roughly one frame in five. Flask, structurally bounded at 60 FPS by `rAF`, still meets the budget. This is the operational regime in which the web stack is necessary rather than optional: a city-scale fleet animation that the QGIS pipeline cannot keep up with on the benchmark hardware. The trade-off is a heavier load phase (a one-shot 26 s wait against ~ 1.4 s for QGIS) and a ~ 700 MB aggregate memory overhead. For an operator-grade live monitoring use case, this trade-off is the price of fluidity at city scale.

Around the crossover ($N \approx 19{,}000$). The narrow band around the crossover is where the verdict is most sensitive to the measurement. At $N = 15{,}000$ the Flask/QGIS ratio is $0.89\times$ in favour of QGIS (QGIS at 67.8 FPS, Flask at 60.0 FPS); at $N = 25{,}890$ the ratio is $1.24\times$ in favour of Flask. Linear-on-log interpolation between these two cells places the crossover at $\approx 19{,}000$ trips, but the exact crossover depends on hardware (CPU, GPU presence, RAM budget), on the QGIS configuration (parallel rendering off, no labelling, single symbol layer), and on the browser version (Chromium 137 in this campaign). A user operating at 15,000 to 20,000 trips on different hardware may find the crossover shifted by a factor of two in either direction, and the practical recommendation in that band is to measure on the actual hardware rather than to extrapolate from this curve.

Reconciliation with Chapter 6. The QGIS contender of this chapter (C6, the memory-layer patch) is the same pipeline characterised in the $N = 5{,}000$ matrix of Section 6.3. The two benchmarks were run with different harnesses (60 s sustained FPS with rendering job in a loop, here; per-frame timing with `renderingFinished` signal, there) and on different STIB captures (*evening peak* 2026-04-24 for the six-architecture matrix; *daytime to evening* 2026-05-02 here). The C6 reading at $N = 5{,}000$ in this chapter (157 FPS) is therefore not numerically identical to the $N = 5{,}000$ reading in Chapter 6 (134 FPS), but the two figures are within the same order of magnitude and the qualitative ordering of the conclusions is consistent: C6 is the deployable upstream patch, fast on sparse-geometry workloads, expression-engine-bounded as N grows. The web stack of this chapter is exercised separately at higher N ranges than the six-architecture matrix and is the necessary complement above the crossover.

8.7 Limitations

Four limits frame the operational reading of this chapter.

(1) Hardware envelope. All measurements run on the single Linux VM of Section 5.1 (4 vCPU, 7.2 GiB RAM, no discrete GPU). The crossover at $N \approx 19{,}000$ is a function of that hardware. A CPU upgrade would push the QGIS curve right (higher rendering throughput, later crossover); a GPU-accelerated Chromium would either lift the Flask cap towards the actual screen refresh (120 FPS on capable monitors) or simply tighten the constancy of the 60 FPS reading, but would not move Flask off the cap. Cross-hardware portability is in the future-work agenda (Section 11).

(2) Workload envelope. The benchmark is run on STIB-shaped trips ($\sim$ 14 vertices per trip on average, 5 s temporal sampling, urban routes); the geometric-complexity analysis of Section 6.6 showed that the QGIS expression engine pays a steep cross-dataset slope on denser geometries (AIS-shaped trips, $\sim$ 1,230 vertices per trip on average). The current chapter does not repeat the AIS comparison at the larger N values used here; a dense-geometry cell at $N = 15{,}000$ AIS is in the future-work agenda. The operational conclusions of Section 8.6 therefore apply to STIB-shaped urban workloads; AIS-shaped maritime workloads at city scale would shift the crossover towards smaller N.

(3) Web-stack envelope. The web contender is a single stack (Flask + MapLibre WebGL). Other modern web stacks (deck.gl [42], kepler.gl [43], MovingPandas-Explorer [53]) use different rendering strategies (instanced WebGL particle systems, Mapbox-GL forks, server-side HTML widgets) and would shift the crossover in either direction. Adding a deck.gl cell at $N = 15{,}000$ and $N = 25{,}890$ on identical NDJSON samples is the natural extension and is listed as future work.

(4) FPS commensurability. The Flask FPS is the cadence of `requestAnimationFrame` callbacks in the browser, structurally capped at 60 Hz by design. The QGIS FPS is the rendering throughput of `QgsMapRendererSequentialJob` cycles run outside the Qt event loop, with no cap. The cross-stack speedup ratio of Section 8.5 below parity is therefore informative about the headroom of the QGIS pipeline over the 60 FPS budget rather than about absolute frame throughput equality. The discussion frames the comparison as a question about *whether each pipeline meets the screen-refresh budget at the working-set size of the user*, not as a multiplicative supremacy claim. A tighter follow-up benchmark would tag every position update on the browser side and report a strictly commensurable "distinct positions painted per second" metric against the same on QGIS; this measurement is listed in future work.

Chapter 9

Discussion

The preceding chapters were descriptive (Sections 1–5), constructive (Sections 6–7) or comparative (Section 8). This short chapter steps back and asks four broader questions. How does the work generalise beyond Brussels (§9.1)? How does it compare to parallel MobilityDB directions (§9.2)? What are the ethical and operational responsibilities of running RTDataHub on real-time transit data (§9.3)? And what would it take to turn the qualitative case studies of Chapter 7 into a measured false-positive rate (§9.4)?

9.1 Generalising Beyond Brussels

Every evaluation uses Brussels feeds (STIB [7], De Lijn [8]) or Manzer's canonical Danish AIS dataset [2]. Each contribution still carries over. The column-oriented refactoring (§6) relies only on `tsample` and vectorised NumPy slicing. These are available in any MobilityDB installation and are bounded only by the $N \times T$ memory cost. The three-layer RTDataHub (§7.3) isolates upstream heterogeneity (different feed formats) in the ingestion layer. Adding TEC or NMBS/SNCB requires one more ingestion module, and the transform/load/API layers stay unchanged, matching the Transitland-style aggregator pattern [54]. The bunching detector's Belgian-calendar context (Meeus's Easter [69]) is a country-specific plug-in; the frequency-aware threshold itself is portable and consistent with the operational literature [52]. The web stack (Flask, MapLibre, MVT) is standard, and Brussels-specific frontend details are isolated in a small config file.

9.2 Comparison with Parallel Directions in the MobilityDB Community

The MOBDB community works on the engine itself (operators, indexes, types), on tooling (PyMEOS, MovingPandas integrations [53]) and on application case studies. This work contributes to the last category and secondarily to the tooling track (the refactored MOVE plugin); it does not extend the core engine. Where we depart from most case studies is the emphasis on *operational continuity*. Most case studies are batch analyses of historical data, whereas RTDataHub has run continuously on the live STIB and De Lijn feeds since mid-April 2026 — over five weeks at the time of writing — reconstructing in that span roughly 2.1 million STIB trips and 115,000 De Lijn trips. The engineering lessons around graceful degradation that this exposed are under-represented in academic writing.

A reader may ask why RTDataHub was built at all, given the regional Mobility Twin Brussels initiative [56]. The two are not substitutes. Mobility Twin is a broad institutional aggregator

that spans operators and mobility modes; RTDataHub is a focused, MOBDB-native platform for public-transport trajectories, restricted to the STIB-MIVB and De Lijn feeds and released as open, reproducible code (Section 11.5). The state of the art (Section 4) already frames it as one concrete, single-operator-class implementation of that vision rather than a competitor to it. The contribution here is not a rival aggregator but a demonstration that the storage-to-visualisation path for live transit trajectories can be closed end to end on a single MOBDB store; that path is precisely the part a wider digital twin would still have to build, and the engineering lessons of this work transfer to it directly.

9.3 Ethical and Operational Responsibilities

A platform that displays the live position of every bus and tram in a city raises a small number of ethical and operational questions. This section addresses them in turn.

Data-protection framing (GDPR). The STIB [7] and De Lijn [8] feeds are public, anonymous at the vehicle level (vehicle IDs rotate between signings-on), and licensed for open reuse. No PII (personally identifiable information) about passengers or drivers is ingested. Under the GDPR [70], the data falls outside the definition of *personal data* (Article 4(1)) since it cannot reasonably be linked to an identifiable natural person. This is consistent with Belgian open-mobility guidance [55], and it is why RTDataHub runs without consent flows or a DPIA (Data Protection Impact Assessment). A future ingestion of personal data (e.g. a ticketing feed) would require a separate GDPR assessment *before* enabling ingestion, with per-feed pseudonymisation in the ingestion layer rather than downstream.

Misuse avoidance. The bus-bunching detector produces alerts that, if misread as driver-level KPIs, could be misused. The platform frames alerts as line-level events keyed on (line, direction, time); no driver attribution is attempted. This matches Cats [52] and is enforced at the API layer: no endpoint exposes any driver-identifying field.

Retention policy. RTDataHub polls continuously and stores the output. Upstream rate limits are respected (§7.4, §7.6). The current prototype does *not* enforce automated retention: raw observations and derived alerts are append-only, and pruning is manual. For a production deployment, the proposed policy is:

- *Raw position observations:* 30 days hot retention, then either purge or archive to cold storage in pseudonymised form (vehicle identifiers replaced by a per-day rotating salt). The 30-day window is enough to support week-over-week analysis and incident replay.
- *Reconstructed trips* (`rt.<feed>_trip`): 12 months retention, then aggregation to weekly / monthly summaries; balances reproducibility (re-runs of the bunching analysis on a recent quarter) against GDPR Article 5(1)(c) (data minimisation).
- *Alerts* (`rt.<feed>_alert`): kept indefinitely as line-level events keyed on (*line*, *direction*, *time*), since they carry no driver-level or vehicle-level identifier in the schema (Appendix A).

The pruning job is a single `psql` cron entry per table; a production deployment would add it before opening the platform to external users.

Re-identification risk. The position feeds are public and anonymous at the vehicle level by design, per-day rotating identifiers, no driver attribution. We nevertheless acknowledge a residual re-identification risk. An external party with access to an internal STIB-MIVB duty-roster could in principle correlate a $(line, direction, timestamp)$ tuple with the on-duty driver at that moment. Two mitigations apply: (i) the platform *never* ingests duty-roster data, so the join cannot be performed inside the system; (ii) alerts are stored at line-direction-time granularity rather than per-vehicle, which removes the simplest correlation key. A production deployment with regulatory clearance would add an explicit access-control layer at the API tier.

API access and CORS. The Flask API runs *same-origin*: frontend and API are served from the same host, with no CORS (cross-origin resource sharing) middleware. Cross-origin embedding would need a one-line header injection, not currently in scope. The API exposes `GET` endpoints only; ingestion and configuration run on the internal Python process that owns the DB connection. The prototype has no authentication and no rate-limiting; the only access control is the network bind, since the server listens on `127.0.0.1` alone and is therefore reachable from the deployment host rather than the public internet. A `GET`-only, same-origin, loopback-bound endpoint is not a substitute for authentication: an explicit access-control layer would be required before exposing the API beyond the host.

These details separate a research prototype from a deployable system.

9.4 From Qualitative Case Studies to a Validated Detector

The proximity, slow-speed and stuck detectors are evaluated qualitatively in the present work (Case Study 1, Case Study 2 and the diurnal-rate analysis of §7.13.6). This section discusses the conditions under which those qualitative results would become a measured false-positive rate, and proposes a concrete protocol ready for a follow-up campaign. Reframing the open question as an executable protocol is itself part of the discussion: it clarifies what the present work does and does not establish, and what a downstream research effort would need.

The protocol is:

1. *Sample.* Draw a stratified sample of ≥ 50 alerts per detector from a single weekday-peak window (e.g. Tuesday 07:00–09:00), stratified equally over `HIGH` / `MEDIUM` / `LOW` severities. Smaller samples suffice for an order-of-magnitude estimate of false-positive rate; ≥ 50 per severity gives a 95% Wilson confidence interval no wider than ± 14 percentage points.

2. *Annotate.* For each alert, replay the relevant window in the RTDataHub frontend and record a binary judgement *true positive / false positive* per the operational criterion (e.g. for proximity: "the two trips were genuinely within the displayed gap, on the same line, and not at a terminal layover"). Inter-annotator agreement on a 20-alert pilot subset would calibrate the criterion before full sampling.

3. *Estimate precision.* Report $\text{precision}_s = \text{TP}_s/(\text{TP}_s + \text{FP}_s)$ per severity s, with a Wilson 95% confidence interval.

4. *Estimate recall (harder).* Recall requires a sample of *ground-truth* bunching events independent of the detector. Two approximate sources are available: (a) visual scrubbing of the RTDataHub frontend over the same window with the alert layer hidden, and (b) retrospective audits of the STIB-MIVB operational disruption log for the same date. Either gives a partial denominator and a partial recall figure; a strict recall would require an external dataset of confirmed bunching events that this work does not have access to.

5. *Sensitivity sweep.* Re-emit the alert stream over the same window with thresholds at $\{3\%, 5\%, 7\%\}\times$ nominal spacing for `HIGH` (and analogously for the other two tiers); plot precision against fraction. A precision-vs-fraction curve identifies the operating point at which the false-positive rate crosses an actionable threshold.

The protocol is straightforward; the gating factor is annotator time ($\sim$ 1 working day for $50 \times 3 = 150$ alerts at ~ 3 min each), not infrastructure.

Chapter 10

Conclusion

This work rebuilt the MobilityDB-to-QGIS visualisation pipeline so it scales to city-scale mobility datasets without the memory and latency collapses documented by Manzer [2]. The work proceeded along two parallel tracks, each answering part of the research agenda of Section 1.3.

10.1 Summary of the Findings

On the QGIS track (RQ1). Chapter 6 compared six rendering pipelines on two contrasting datasets (∼5 000 trips each, sparse STIB at ∼14 vertices/trip and dense AIS at ∼1 230 vertices/trip) and identified three deployment tiers. The latency leader is a targeted optimisation of Manzer's published canvas-item architecture [2] (C2: ∼181 FPS on STIB, ∼65 FPS on AIS, i.e. 21.8× / 35.7× over the MOVE upstream baseline), the only condition to clear the 60 FPS budget on both datasets. The deployable upstream contribution is a candidate opt-in memory-layer patch to MOVE [62] (C6: 16.1× on STIB, 4.4× on AIS); the asymmetry is consistent with the architectural hypothesis that the QGIS expression engine, untouched by the patch, remains the per-frame bottleneck on dense geometries. The analytical option is a column-oriented MOBDB pre-sampling into a NumPy matrix (C4: 35 FPS on AIS at ∼760 MB RAM, 44 FPS on STIB at ∼890 MB RAM), suitable for offline analysis but disqualified as a plugin default by the memory cost. The pattern matches the progressive-visualisation principle of Ulmer et al. [45]; the residual cross-dataset slope on the expression-engine pipelines motivates the cross-stack comparison of Chapter 7.

On the web track (RQ2/RQ3). Chapter 7 introduced **RTDataHub**, a three-layer platform. It ingests the stateless STIB feed [7] and the GTFS-Realtime De Lijn feed [4, 8], and normalises them into a single MobilityDB store [1]. It then reconstructs trajectories by projecting them onto GTFS shapes [10] (a lightweight linear-referencing scheme; the heavier HMM map-matchers of [38] are not invoked because the feeds are already odometric or carry a stable trip identifier). It classifies every observation against a Belgian calendar via Meeus's Easter algorithm [69], and exposes the result through a vector-tile API [14, 15] to a minimal MapLibre frontend [6]. The same platform hosts an automated bus-bunching detector whose thresholds adapt to the planned frequency at the current time of day, following Cats [52] on the classical foundation of [17]. A clamped fallback chain keeps it functional when upstream feeds degrade.

Measured end-to-end latency is ≈10 s median for STIB (p95 ≈24 s) and ≈11 s median for De Lijn (p95 ≈25 s), well under any reasonable "real-time" threshold [61]. Time-to-First-Pixel is the API's design constraint, targeted at the 500 ms threshold of Liu and Heer [24]; a direct

measurement on canonical pan / zoom / time-seek interactions is part of the follow-up agenda (§7.15).

10.2 Complementarity of the Two Tracks

The head-to-head comparison of Section 8, run on nine scales from $N = 100$ to $N = 25{,}890$ simultaneous trips (the latter being the full live STIB working set on the chosen window), draws the boundary at an interpolated $N_{\text{cross}} \approx 19{,}000$ trips. The QGIS contender of this comparison is the candidate MOVE upstream patch (C6 of Chapter 6), not the column-oriented variant explored as an analytical option. Below the crossover, QGIS C6 sustains 486 FPS at $N = 100$ down to 68 FPS at $N = 15{,}000$: every cell exceeds the 60 FPS browser `requestAnimationFrame` cap that bounds the Flask side. The two pipelines therefore deliver the same perceived frame rate at every cell below the crossover, on a 60 Hz display, while the QGIS pipeline carries 1.5× to 8× of headroom over the cap. Aggregated RAM is strictly higher on the Flask side at every cell (+162 MB at $N = 100$ to +693 MB at *all_day*), reflecting the cost of carrying a transport layer (JSON or NDJSON), a separate renderer process (Chromium), and a PostgreSQL instance alongside the rendering pipeline. The load phase is 5–20× shorter on the QGIS side because the data does not leave the process.

The QGIS track is the right answer when the user is a domain analyst at a desktop with a specific spatial-analytic question and wants the full GIS environment (arbitrary PostGIS predicates, ad-hoc overlays, direct MobilityDB operators), or when the working set sits below the crossover. The candidate MOVE upstream patch of Chapter 6 preserves the public API of the plugin and is therefore directly deployable to existing MOVE users at this regime.

The web track becomes necessary above the crossover: at $N = 25{,}890$ (the full live STIB working set on the chosen window), QGIS C6 sustains 48 FPS against the 60 FPS budget that Flask still meets. The user perceives the difference. The thin-client architecture — heavy work server-side, GPU-friendly rendering client-side — follows Johnson et al. [25] and Tremmel and Zink [26], and is the only one of the two pipelines we observed sustaining the screen-refresh budget at the full Brussels network size on this hardware. The cost is a one-shot 26 s load phase against 1.4 s for QGIS, and an aggregated RAM footprint of 1,424 MB against 731 MB. A comparison against deck.gl, kepler.gl or MovingPandas-Explorer baselines is left for future work.

10.3 Lessons Learned

Four broader lessons go beyond the Brussels case.

(1) Move the expensive work to the right layer, but watch where the new bottleneck lands. The C2 latency leadership of Section 6 (21.8× STIB, 35.7× AIS over the MOVE upstream baseline) came not from a new algorithm but from removing three Python-side hot spots from Manzer's published canvas-item architecture. The C6 patch separately removed the provider call but left the QGIS expression engine in place; on dense geometries the engine becomes the new ceiling — breaking that one is no longer an intra-stack question, and motivates the web track of Chapter 7.

(2) Heterogeneous feeds need a single source of truth. Two neighbouring operators (STIB-MIVB and De Lijn) disagree on identifiers, cadence and error modes. A three-layer ingest→transform→load architecture with MOBDB as the single source of truth absorbs the heterogeneity in one place.

(3) Graceful degradation is architectural. The case studies of Section 7.13 all involved upstream data problems. The platform kept functioning only because clamped fallback and

freshness gates were designed in from day one; retrofitting them is much harder.

(4) No single visualisation paradigm serves all users. GIS-skilled analysts benefit from the QGIS environment; operators and non-technical stakeholders benefit from the zero-install browser view. “Desktop or web?” is underspecified until one also asks “which user, which task”.

Chapter 11

Future Work

The contributions of this thesis open several concrete avenues for follow-up work. They are organised below into five families: desktop extensions of the QGIS track (RQ1), web extensions of the RTDataHub track (RQ2/RQ3), broader Brussels-perimeter coverage, methodological strengthening of the validation protocols, and broader research directions.

11.1 QGIS track (RQ1)

Three directions follow from the column-oriented pattern. *Tiled pre-sampling* would partition the time axis into hourly tiles with a rolling-window memory budget, decoupling the matrix size from the full horizon [45]. *A bulk QGIS geometry API* would push a full column slice as one contiguous buffer, removing the residual `changeGeometryValues` ceiling. The same pattern would also speed up temporal joins, density maps and vector-field rendering, generalising the plugin from a visualisation component into a broader column-oriented MOBDB accelerator.

Three further directions target the MOVE plugin itself. The C6 patch removed the PostgreSQL provider call but left the QGIS expression engine untouched; a *combined provider-and-expression patch* that pairs the opt-in memory layer with a direct per-frame geometry write, bypassing the expression evaluation entirely, would carry the C2 latency profile into the public plugin API — which C2 itself cannot, since it bypasses that API. *A GPU-backed symbol layer*, together with re-enabling the QGIS parallel renderer that was switched off for every benchmarked condition on the GPU-less measurement machine, would isolate how much of the residual per-frame cost is raster render rather than compute. *A temporal index on the in-memory layer* — an active-trip filter keyed on the current animation window — would let the renderer skip trips outside the visible interval, the memory-layer analogue of the temporal pre-filter that drove the C2 speedup.

11.2 Web track (RQ2/RQ3)

The hot/cold loader split (Section 7.8) transfers wholesale to De Lijn. A migration to MLT [26] would roughly halve payload size on attribute-heavy tiles. Reducing *false positives near terminals* via `stops.txt` annotation is listed in Section 7.11. The detector can also be extended to *headway gaps*, *run-abandonment* and *delay cascades*, each maps neatly onto a MOBDB query. The accumulated alert archive is itself a dataset for *predictive bunching*.

On the frontend, the MapLibre symbol layer currently sustains only a few thousand simultaneously animated vehicles. Replacing it with a *deck.gl overlay* — a `ScatterplotLayer` and `PathLayer` for vehicles and trajectories, a `TripsLayer` for temporal playback, interleaved with the existing vector-tile base map through a `MapboxOverlay` so the alert layers and map interactions

are preserved — would move the per-frame work onto GPU-instanced rendering and lift that ceiling into the tens of thousands. deck.gl would then join QGIS C6 and the Flask + MapLibre stack as a third contender in the head-to-head protocol of Chapter 8. Two caveats temper the expected gain: the browser `requestAnimationFrame` cap of 60 FPS still bounds the frame rate, so the benefit is rendering headroom and stability rather than a higher number; and a build step would return, eroding the no-bundler simplicity of the present frontend.

A more concrete set of six follow-ups falls out of the chapter. *Native MOBDB extension.* Prototyping a server-side C function that exposes the pre-sampled column matrix $(\mathbf{X}, \mathbf{Y})$ straight to a thin client over a custom binary protocol would short-circuit the Python loop entirely; this is the natural extension of the bottleneck-shift insight of Chapter 6. *In-production shadow-run of `hungarian_merge` vs. `greedy_chainmerge`.* The seven-day offline benchmark of Section 7.5.3 already places `greedy_chainmerge` ahead of `hungarian_merge` on coverage and fragmentation, but it cannot observe the two matchers under the live streaming cadence. Running both in parallel against the same multi-day production stream, with a manually annotated sample of ∼50 stratified fusions for precision in a Wilson interval, would quantify the empirical cost of dropping the GTFS-anchored condition under production conditions and either confirm or revise the engineering choice. *Dedicated systemd timer for the chain-merge stage.* The hourly chain-merge is currently triggered from inside the main pipeline runner via `STIB_CHAIN_MERGE_INTERVAL_S`; decoupling it into a stand-alone `systemd` timer would isolate its scheduling from the streaming matcher and simplify the operational story. *Re-canonicalisation of the historical merge debt.* About 6.8% of pre-P1 fusions point to a root that is not chronologically first in its chain; a one-shot SQL script (provided in the companion repository) can re-root these ∼1 476 chains in ∼3 s wall without disturbing the merged `tgeompoint` geometry. *Off-route HMM map-matcher.* Integrating Newson & Krumm [38] into the transformer would let the platform keep tracking a vehicle that deviates from its planned shape (incident, detour, manual short turn), a case the current linear-referencing scheme silently mishandles. The cost is calibration of the HMM emission model and one new transform stage. *Parquet export to DuckDB analytics.* Monthly partitioned export of `rt.stib_trip` and `rt.delijn_trip` to Parquet [71], queryable from DuckDB [72] without touching the live database, would extend analytical access beyond the live retention window of Section 9.3. The schema is already columnar-friendly (one row per trip, one `tgeompoint` per row).

A C++ backend on Wt [73] with direct MEOS integration would remove the Python↔C boundary on every hot-path call. A more radical direction, suggested during the supervision of this work, is to compile MEOS itself to WebAssembly (portable binary instruction format for browsers) (WASM) and run the temporal-geometry operators *inside the browser*: the Flask + MapLibre split would collapse into a single client-side stack with direct, no-network access to the same C operators that the server uses today, removing both the JSON serialisation cost and the Python wrapper overhead in one move. This would also reopen the trade-off space between server-side rendering (current architecture) and on-device analytical queries on a still-warm trajectory cache.

11.3 Brussels perimeter

Adding TEC (Walloon buses) and NMBS/SNCB (national rail) would yield a unified multi-modal picture of the region; the cost is one ingestion adapter per operator, the schema is unchanged. Connecting RTDataHub to the Mobility Twin Brussels initiative [56] would extend institutional reach. A proper HMM map-matcher [38] would handle off-route diversions silently

mishandled by linear referencing.

11.4 Validation and methodology

Several headline numbers would be strengthened by measurement rather than by new engineering. The FPS and latency figures of Chapters 6 and 8 are *point estimates*: repeating each run and bootstrapping over hourly slices would attach proper confidence intervals to the reported speedups and to the N_{cross} crossover. The bus-bunching detector is so far validated only qualitatively, through the case studies of Section 7.13; executing the precision/recall protocol already specified in Section 9.4 — a stratified annotated sample scored with Wilson intervals — would turn a demonstrated detector into a measured one. The camera ground-truth of Section 7.5.7 rests on a single afternoon and 125 observations; a multi-day, multi-camera campaign, ideally in a second city, would separate matcher quality from session-specific collisions. And the *which user, which task* principle stated among the Lessons Learned above is argued but never tested: a controlled study with transit-operations staff and GIS analysts, on representative incident-triage and spatial-query tasks, would put it on an empirical footing.

11.5 Broader research

The pre-sampled column matrix is the right shape for *vector-field visualisation* [50]: derivatives give velocities, binning gives the grid. WebGL particle systems would lift the client-side ceiling beyond the few thousand vehicles MapLibre currently sustains. *Cross-city benchmarking* (Paris, London, Amsterdam) would test how well the contributions generalise.

Data and Code Availability

The source code for the refactored QGIS MOVE plugin (Section 6) and for the RTDataHub platform (Section 7) is released under an open-source licence and can be consulted at the two repositories below.

- **Refactored QGIS MOVE plugin and RQ1 benchmarks**: `https://github.com/Aelhamri/move-bench` — the plugin source together with the six-architecture benchmark scripts and the raw per-run dumps behind Section 6.
- **RTDataHub platform**: `https://github.com/Aelhamri/LocalRtdatahub` — a simplified, self-contained version of the platform (ingestion, transform, API, frontend) with the benchmark and case-study scripts of Section 7. The full production deployment cannot be released in its entirety; this repository carries the reproducible subset.

The two reference datasets used in this thesis are themselves publicly available:

- The **Danish AIS dataset** [2] used for the RQ1 baseline benchmark is archived at the Danish Maritime Authority open-data portal.
- The **STIB-MIVB** [7] and **De Lijn** [8] feeds used by RTDataHub are consumed at runtime from the operators' public APIs; a snapshot of the captures used for the three Case Studies of Section 7.13 is preserved in the RTDataHub repository above.

The pinned software versions used for the measurements reported in the present work are:

- PostgreSQL 17.9 with PostGIS 3.5.2
- MobilityDB 1.4.0 (built against PostGIS 3.5)
- PyMEOS 1.3.x
- QGIS 3.34 LTR
- Python 3.13.5 with NumPy 2.4.4, pandas 3.0.2, Shapely 2.1.2, GeoPandas 1.1.3
- Flask 3.1.3, MapLibre GL JS 4.x

A top-level `requirements.txt` in the RTDataHub repository pins the Python dependencies of the server side; the MOVE plugin ships its own `plugin/requirements.txt` for the Python dependencies required inside QGIS. The database side (PostgreSQL, PostGIS, MobilityDB) is assumed to be provisioned separately at the versions listed above; a unified `docker-compose.yml` is not shipped with the current prototype and is flagged as future work (Section 11), because the ingestion workers, the Flask API and the browser frontend can all be started from the same Python interpreter during development and do not require container orchestration to reproduce the numbers in this work.

Declaration of Generative AI and AI-assisted Technologies in the Writing Process

During the preparation of this work, the author used Gemini (Google) as an LLM-based writing assistant in order to obtain editorial suggestions on phrasing, paragraph structure and consistency across chapters. After using these tools, the author reviewed and edited the content as needed and takes full responsibility for the content of this thesis.

Grammarly was additionally used as a spelling and grammar checker. Under the present guidelines, basic spelling- and grammar-checking tools do not require declaration; it is noted here only for completeness.

No generative AI tool was used to produce results, data, figures or code: every measurement, every figure-generation script, every line of source code, every algorithmic design choice and every benchmark in this thesis is the author's own, except where explicitly attributed to the cited upstream projects and to the external contribution of Dawid Banas recorded in the Disclaimer of Section 1.7.

Bibliography

[1] E. Zimányi, M. Sakr, and A. Lesuisse, "MobilityDB: A mobility database based on PostgreSQL and PostGIS," *ACM Transactions on Database Systems*, vol. 45, no. 4, pp. 1–42, 2020.

[2] A. I. Manzer, "Visualization of MobilityDB data in QGIS," master's thesis, Université Libre de Bruxelles, 2024. Previous work serving as the baseline for this research.

[3] R. H. Güting and M. Schneider, *Moving Objects Databases*. Morgan Kaufmann, 2005. Seminal monograph on moving object data models.

[4] Google, "General transit feed specification realtime (GTFS-RT) reference." https://gtfs.org/realtime/reference/, 2024. Accessed: May 4, 2026.

[5] QGIS Development Team, "QGIS geographic information system." https://qgis.org, 2024. Accessed: May 4, 2026.

[6] MapLibre Contributors, "MapLibre GL JS: Open-source JavaScript library for embedded maps." https://maplibre.org, 2024. Accessed: May 4, 2026.

[7] STIB-MIVB, "Open data portal." https://opendata.stib-mivb.be, 2024. Accessed: May 4, 2026.

[8] De Lijn, "GTFS-Realtime feed." https://www.delijn.be/en/overdelijn/organisatie/zoekdocument/opendata.html, 2024. Accessed: May 4, 2026.

[9] B. Tversky, J. B. Morrison, and M. Betrancourt, "Animation: can it facilitate?," *International Journal of Human-Computer Studies*, vol. 57, no. 4, pp. 247–262, 2002.

[10] Google, "General transit feed specification (GTFS) reference." https://gtfs.org/schedule/reference/, 2024. Accessed: May 4, 2026.

[11] A. Antrim and S. J. Barbeau, "The many uses of GTFS data – opening the door to transit and multimodal applications," tech. rep., Location-Aware Information Systems Laboratory, University of South Florida, 2013.

[12] PostGIS Development Team, "PostGIS: Spatial and geographic objects for PostgreSQL." https://postgis.net, 2024. Accessed: May 4, 2026.

[13] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, (Boston, MA, USA), pp. 47–57, 1984.

[14] Mapbox, "Mapbox vector tile specification." https://github.com/mapbox/vector-tile-spec, 2021. Accessed: May 4, 2026.

[15] Pallets Projects, "Flask: A lightweight WSGI web application framework." https://flask.palletsprojects.com, 2024. Accessed: May 4, 2026.

[16] G. F. Newell and R. B. Potts, "Maintaining a bus schedule," in *Proceedings of the 2nd Australian Road Research Board Conference*, vol. 2, pp. 388–393, 1964.

[17] C. F. Daganzo, "A headway-based approach to eliminate bus bunching: Systematic analysis and comparisons," *Transportation Research Part B: Methodological*, vol. 43, no. 10, pp. 913–921, 2009.

[18] X. J. Eberlein, N. H. M. Wilson, and D. Bernstein, "The holding problem with real-time information available," *Transportation Science*, vol. 35, no. 1, pp. 1–18, 2001.

[19] J. J. Bartholdi and D. D. Eisenstein, "A self-coordinating bus route to resist bus bunching," *Transportation Research Part B: Methodological*, vol. 46, no. 4, pp. 481–491, 2012.

[20] L. Moreira-Matias, O. Cats, J. a. Gama, J. a. Mendes-Moreira, and J. F. de Sousa, "An online learning approach to eliminate bus bunching in real-time," *Applied Soft Computing*, vol. 47, pp. 460–482, 2016.

[21] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[22] The pandas development team, "pandas-dev/pandas: Pandas." https://pandas.pydata.org, 2024. Accessed: May 4, 2026.

[23] M. Bakli, M. Sakr, and E. Zimányi, "PyMEOS: Python bindings for the Mobility Engine Open Source." Open-source software, https://github.com/MobilityDB/PyMEOS, 2023. Accessed: May 4, 2026.

[24] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2122–2131, 2014.

[25] G. Johnson, S. Molnar, N. Brunhart-Lupo, and K. Gruchalla, "Architecture for web-based visualization of large-scale energy domains," in *2024 IEEE Workshop on Energy Data Visualization (EnergyVis)*, pp. 46–51, October 2024.

[26] M. Tremmel and R. Zink, "MapLibre Tile: A next generation vector tile format," in *Proceedings of the 33rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pp. 1118–1121, 2025.

[27] M. A. Quddus, W. Y. Ochieng, and R. B. Noland, "Current map-matching algorithms for transport applications: State-of-the-art and future research directions," *Transportation Research Part C: Emerging Technologies*, vol. 15, no. 5, pp. 312–328, 2007.

[28] M. Sakr and R. H. Güting, "Spatiotemporal pattern queries," *GeoInformatica*, vol. 15, no. 3, pp. 497–540, 2011.

[29] N. Pelekis and Y. Theodoridis, *Mobility Data Management and Exploration*. New York, NY: Springer, 2014.

[30] E. Zimányi, M. Sakr, and A. Lesuisse, *Mobility Data Science: From Data to Insights*. Springer, 2023. Reference textbook for the MobilityDB ecosystem.

[31] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-Tree: A dynamic index for multi-dimensional objects," in *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB)*, (Brighton, UK), pp. 507–518, 1987.

[32] N. Pelekis, E. Frentzos, N. Giatrakos, and Y. Theodoridis, "HERMES: Aggregative LBS via a trajectory DB engine," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ACM, 2008.

[33] M. Bakli, M. Sakr, and T. H. A. Soliman, "HadoopTrajectory: A Hadoop spatiotemporal data processing extension," *Journal of Geographical Systems*, vol. 21, pp. 211–235, 2019.

[34] Z. Zhang, C. Jin, J. Mao, X. Yang, and A. Zhou, "TrajSpark: A scalable and efficient in-memory management system for big trajectory data," in *Web and Big Data (APWeb-WAIM 2017)*, vol. 10366 of *Lecture Notes in Computer Science*, Springer, 2017.

[35] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel approaches in query processing for moving object trajectories," in *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, (Cairo, Egypt), pp. 395–406, 2000.

[36] M. Hojati, S. Roberts, and C. Robertson, "DSTree: A spatio-temporal indexing data structure for distributed networks," *Mathematical and Computational Applications*, vol. 29, no. 3, p. 42, 2024.

[37] X. Zhao, K.-Y. Lam, and T.-W. Kuo, "Indexing spatiotemporal trajectory data streams on key-value storage," *Computing*, vol. 106, no. 8, pp. 2707–2735, 2024.

[38] P. Newson and J. Krumm, "Hidden Markov map matching through noise and sparseness," in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, (Seattle, WA, USA), pp. 336–343, 2009.

[39] B. Yang, Y. Zhang, W. Lu, and L. Wei, "Map-matching on big data: a distributed and efficient algorithm with Hidden Markov Model," *IEEE Access*, vol. 6, pp. 64296–64308, 2018.

[40] R. Scheepens, N. Willems, H. van de Wetering, G. Andrienko, N. Andrienko, and J. J. van Wijk, "Composite density maps for multivariate trajectories," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2518–2527, 2011.

[41] V. Agafonkin, "Leaflet: An open-source JavaScript library for mobile-friendly interactive maps." https://leafletjs.com, 2024. Accessed: May 4, 2026.

[42] Uber Technologies, "deck.gl: WebGL2-powered framework for visual exploratory data analysis of large datasets." https://deck.gl, 2024. Accessed: May 4, 2026.

[43] S. He, "From beautiful maps to actionable insights: Introducing kepler.gl." Uber Engineering Blog, https://eng.uber.com/keplergl/, 2018. Accessed: May 4, 2026.

[44] G. Andrienko, N. Andrienko, P. Bak, D. Keim, and S. Wrobel, *Visual Analytics of Movement*. Springer, 2013.

[45] A. Ulmer, M. Angelini, J.-D. Fekete, J. Kohlhammer, and T. May, "A survey on progressive visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 30, no. 9, pp. 6447–6467, 2024.

[46] J.-D. Fekete and R. Primet, "Progressive analytics: A computation paradigm for exploratory data analysis." arXiv:1607.05162, 2016. Accessed: May 4, 2026.

[47] S. K. Card, G. G. Robertson, and J. D. Mackinlay, "The information visualizer, an information workspace," in *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*, (New Orleans, LA, USA), pp. 181–186, 1991.

[48] W. R. Tobler, "A computer movie simulating urban growth in the detroit region," *Economic Geography*, vol. 46, pp. 234–240, 1970.

[49] G. Andrienko and N. Andrienko, "Designing visual analytics methods for massive collections of movement data," *Cartographica*, vol. 42, no. 2, pp. 117–138, 2007.

[50] A. Li, Z. Xu, J. Zhang, T. Li, X. Cheng, and C. Hu, "A vector field visualization method for trajectory big data," *ISPRS International Journal of Geo-Information*, vol. 12, no. 10, p. 398, 2023.

[51] W. Feng and M. A. Figliozzi, "Empirical findings of bus bunching distributions and attributes using archived AVL/APC bus data," in *Proceedings of the International Conference on Civil and Transportation Engineering (ICCTP 2011)*, pp. 4330–4341, 2011.

[52] O. Cats, "Regularity-driven bus operation: Principles, implementation and business models," *Transport Policy*, vol. 36, pp. 223–230, 2014.

[53] A. Graser, "MovingPandas: Efficient structures for movement data in Python," *GI_Forum*, vol. 1, pp. 54–68, 2019.

[54] Interline Technologies, "Transitland: An aggregated open transit data platform." https://www.transit.land, 2019. Accessed: May 4, 2026.

[55] SPF Mobilité et Transports, "Enquête monitor sur la mobilité en belgique," tech. rep., Government of Belgium, 2023.

[56] Mobility Twin Project, "Mobility Twin Brussels: A digital twin for urban mobility." https://mobilitytwin.brussels, 2024. Accessed: May 4, 2026.

[57] L. Merten, S. Stinckwich, *et al.*, "Brussels Mobility Twin: Architecture and use cases," in *Proceedings of the 31st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '23)*, 2023.

[58] S. Mihai, M. Yaqoob, D. V. Hung, W. Davis, P. Towakel, M. Raza, M. Karamanoglu, B. Barn, D. Shetve, R. V. Prasad, H. Venkataraman, R. Trestian, and H. X. Nguyen, "Digital twins: A survey on enabling technologies, challenges, trends and future prospects," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 4, pp. 2255–2291, 2022.

[59] T. Hägerstrand, "What about people in regional science?," *Papers of the Regional Science Association*, vol. 24, no. 1, pp. 7–21, 1970.

[60] K. Grossner and K. C. Clarke, "Is GIS a cartographer? dimensioning a time-geographic platform," *Cartographica*, vol. 42, pp. 265–279, 2008.

[61] C. Brakewood and K. Watkins, "A literature review of the passenger benefits of real-time transit information," *Transport Reviews*, vol. 39, no. 3, pp. 327–356, 2019.

[62] M. Schoemans, E. Zimányi, and MOVE contributors, "MOVE: A QGIS plugin for visualising MobilityDB trajectories." Open-source software, https://github.com/MobilityDB/move, 2024. Maintained by the Université libre de Bruxelles. Accessed: May 13, 2026.

[63] R. Jonker and A. Volgenant, "A shortest augmenting path algorithm for dense and sparse linear assignment problems," *Computing*, vol. 38, no. 4, pp. 325–340, 1987.

[64] R. Kimball and M. Ross, *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. Wiley, 3rd ed., 2013.

[65] AITwin, "MobilityTwin.Brussels Components — STIB vehicle identification algorithm." Open-source software (Creative Commons BY-NC-SA 4.0), https://github.com/AITwin/Components, 2026. File `components/stib/harvesters/identify_vehicle/algorithm.py`, vendored from `main` branch on 2026-04-29. Accessed: May 4, 2026.

[66] OpenStreetMap contributors, "OpenStreetMap brussels road network." Open data (ODbL), via the Overpass API, https://overpass-api.de, 2026. Filter `highway=* railway=tram|subway` clipped to the Brussels-Capital bounding box. Accessed: May 4, 2026.

[67] Bruxelles Mobilité, "SMV — spécialisation multimodale des voiries (plan Good Move)." Open data, Region of Brussels-Capital, https://data.mobility.brussels, 2024. Strategic multimodal road classification (`tp_class` flags transit-priority axes). Accessed: May 4, 2026.

[68] Centre d'Informatique pour la Région Bruxelloise (CIRB), "UrbIS-Adm: cartographie administrative de référence de la région de Bruxelles-Capitale." Open data, Region of Brussels-Capital, https://datastore.brussels/web/urbis-download, 2025. Photogrammetric capture at 7.5 cm/pixel; layer `urbadm_ss` (roadway polygons). Accessed: May 4, 2026.

[69] J. Meeus, *Astronomical Algorithms*. Willmann-Bell, 2 ed., 1998.

[70] European Parliament and Council, "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (general data protection regulation)." Official Journal of the European Union, L 119/1, 2016. Accessed: May 4, 2026.

[71] Apache Software Foundation, "Apache Parquet — a columnar storage format for efficient analytics." https://parquet.apache.org, 2013. Accessed: May 4, 2026.

[72] M. Raasveldt and H. Mühleisen, "DuckDB: an embeddable analytical database," in *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*, pp. 1981–1984, ACM, 2019.

[73] Emweb, "Wt: a C++ web toolkit for developing widget-based web applications." https://www.webtoolkit.eu/wt, 2024. Accessed: May 4, 2026.

[74] S. Gillies *et al.*, "Shapely: geometric objects, predicates, and operations for Python." https://shapely.readthedocs.io, 2007. Accessed: May 4, 2026.

Appendix A

SQL Schema of the RTDataHub Tables

This appendix reproduces the `CREATE TABLE` statements for the main tables and views of the RTDataHub MobilityDB store described in Chapter 7. The schema is emitted from Python at bootstrap time by `src/etl/pipeline/load/create_tables.py` in the RTDataHub repository (function `create_tables()`); post-deployment extensions are reported in `data/experimental/replay/sql/migration_up_prod.sql` and re-applied through idempotent `ALTER TABLE ... ADD COLUMN IF NOT EXISTS` clauses inside `create_tables.py` so that a fresh install reaches the same target. The project does not use `pg_dump`. All tables live under the `rt` schema, except the derived view `static.stib_line_thresholds`. The present version reflects the post-refactor state of the store: hot/cold separation of STIB trips (Section 7.8), `greedy_chainmerge` schema additions (Section 7.4.3), and the integration of two editorial-message tables, Traveller Info STIB (Section 7.4.4) and De Lijn GTFS-RT Alert (Section 7.6.1). Indices created alongside the `CREATE TABLE` statements are reproduced; those omitted for readability are noted explicitly.

rt.stib_vehicle_position, raw STIB observations. The flat PostGIS table that stores one row per raw STIB observation from the stateless Open Data Provider feed (Section 7.4). Because the STIB ODP feed does not carry a stable vehicle identifier, `vehicle_uuid` is nullable on this raw table and is only assigned downstream, during trip reconstruction, via proximity matching.

```
CREATE TABLE IF NOT EXISTS rt.stib_vehicle_position (
    position_id         TEXT PRIMARY KEY,
    vehicle_uuid        TEXT,              -- nullable (ODP has no stable vehicle id)
    lineid              TEXT,
    direction           INTEGER,
    pointid             TEXT,
    distance_from_point DOUBLE PRECISION,
    fetched_at          TIMESTAMPTZ NOT NULL,
    geom                GEOMETRY(Point, 4326),
    ingested_at         TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
CREATE INDEX IF NOT EXISTS idx_stib_pos_vehicle ON rt.stib_vehicle_position(vehicle_uuid);
CREATE INDEX IF NOT EXISTS idx_stib_pos_lineid  ON rt.stib_vehicle_position(lineid);
CREATE INDEX IF NOT EXISTS idx_stib_pos_fetched ON rt.stib_vehicle_position(fetched_at);
```

```
CREATE INDEX IF NOT EXISTS idx_stib_pos_geom    ON rt.stib_vehicle_position USING GIST(geom)
```

rt.stib_trip_open / rt.stib_trip_cold / rt.stib_trip: hot/cold separation. The reconstructed STIB trips are split between a hot table `rt.stib_trip_open` that absorbs all `UPDATE`s on trips currently being extended, and an append-only cold archive `rt.stib_trip_cold` that receives only `INSERT`s after graduation. A union-all view `rt.stib_trip` exposes the two tables behind a single read interface to downstream consumers (alert detector, frontend), which remain agnostic to the physical placement of rows. The graduation threshold `TRIP_TIMEOUT_S` is set to 1 800 s (cf. Section 7.4.1); the main loader never writes directly into `stib_trip_cold`, only the graduation worker moves mature rows. The columns `terminus`, `is_deviated` and `daily_trip_seq` exist only on the cold side; they are set at graduation and not modifiable thereafter.

```
CREATE TABLE rt.stib_trip_open (
    trip_id        text PRIMARY KEY,
    vehicle_uuid   text,
    lineid         text NOT NULL,
    direction      integer NOT NULL,
    start_ts       timestamptz NOT NULL,
    end_ts         timestamptz NOT NULL,
    point_count    integer NOT NULL DEFAULT 0,
    line_trip_id   text,
    trip           public.tgeompoint,
    last_point     public.geometry(Point, 4326),
    ingested_at    timestamptz NOT NULL DEFAULT NOW()
);

CREATE TABLE rt.stib_trip_cold (
    trip_id        text PRIMARY KEY,
    vehicle_uuid   text,
    lineid         text NOT NULL,
    direction      integer NOT NULL,
    terminus       text,
    start_ts       timestamptz NOT NULL,
    end_ts         timestamptz NOT NULL,
    point_count    integer NOT NULL DEFAULT 0,
    is_deviated    boolean NOT NULL DEFAULT false,
    trip           public.tgeompoint,
    ingested_at    timestamptz NOT NULL DEFAULT NOW(),
    line_trip_id   text,
    daily_trip_seq integer,
    last_point     public.geometry(Point, 4326)
);

CREATE OR REPLACE VIEW rt.stib_trip AS
  SELECT trip_id, vehicle_uuid, lineid, direction,
         NULL::text AS terminus, start_ts, end_ts, point_count,
```

```
        false AS is_deviated, trip, ingested_at, line_trip_id,
        NULL::integer AS daily_trip_seq, last_point
  FROM rt.stib_trip_open
UNION ALL
SELECT trip_id, vehicle_uuid, lineid, direction,
        terminus, start_ts, end_ts, point_count,
        is_deviated, trip, ingested_at, line_trip_id,
        daily_trip_seq, last_point
  FROM rt.stib_trip_cold
  WHERE superseded_by IS NULL;
```

The geometric point `last_point` is carried by both tables and indexed with GiST to support the `ST_DWithin` query of the SQL-side matcher (Section 7.8). The view `rt.stib_trip` projects the subset of columns shared by the two tables, padding the cold-only columns (`terminus`, `is_deviated`, `daily_trip_seq`) with typed `NULL` literals on the hot side so the `UNION ALL` type-checks. The `WHERE superseded_by IS NULL` clause on both branches is load-bearing: it hides the trip fragments absorbed by the `greedy_chainmerge` chain-merge (Section 7.4.3) from every downstream consumer (alert detector, frontend), so the view exposes only canonical trips. A consumer that needs the full absorbed chain queries `rt.stib_trip_cold` directly. Per-table secondary indices on `lineid`, `vehicle_uuid`, `start_ts`, `end_ts`, `(lineid, daily_trip_seq)`, and `line_trip_id` are created in parallel; they are omitted from the verbatim block above for readability and follow the same naming scheme as the previous `rt.stib_trip` pre-refactor schema.

`greedy_chainmerge` schema additions on `rt.stib_trip_cold`. The `greedy_chainmerge` chain-merge of Section 7.4.3 adds three columns to `rt.stib_trip_cold` that materialise the forest of merged fragments and the idempotence sentinel of the physical fusion, plus a dedicated log table that records every retained edge. These additions are defined in `data/experimental/replay/sql/migration_up_prod.sql` and re-applied through idempotent `ALTER TABLE ... ADD COLUMN IF NOT EXISTS` clauses in `create_tables.py`.

```
ALTER TABLE rt.stib_trip_cold
    ADD COLUMN superseded_by text
        REFERENCES rt.stib_trip_cold(trip_id),
    ADD COLUMN merged_at  timestamptz,
    ADD COLUMN fused_at   timestamptz;

ALTER TABLE rt.stib_trip_cold
    ADD CONSTRAINT stib_trip_cold_no_self_supersede
        CHECK (superseded_by IS NULL OR superseded_by <> trip_id);

CREATE INDEX CONCURRENTLY idx_stib_trip_cold_supby
    ON rt.stib_trip_cold(superseded_by)
    WHERE superseded_by IS NOT NULL;

CREATE INDEX IF NOT EXISTS idx_stib_trip_gist
    ON rt.stib_trip_cold USING GIST(trip);
```

```
CREATE TABLE rt.stib_trip_merge_log (
    merge_id     bigserial PRIMARY KEY,
    kept_id      text NOT NULL,
    absorbed_id  text NOT NULL,
    algo_version text NOT NULL,
    gap_s        double precision,
    dist_m       double precision,
    v_mps        double precision,
    merged_at    timestamptz DEFAULT NOW()
);
CREATE INDEX idx_merge_log_kept     ON rt.stib_trip_merge_log(kept_id);
CREATE INDEX idx_merge_log_absorbed ON rt.stib_trip_merge_log(absorbed_id);
```

The `CHECK` constraint forbids self-loops in the `superseded_by` forest; combined with the `WHERE superseded_by IS NULL` condition applied to the main scan of the chain-merge, it bounds the effective depth of chains to a small residual fraction (typically 0.05% at depth 3, cf. Section 7.5.8). The partial index on `superseded_by` covers only absorbed fragments (a few per cent of the row population) and supports the `root.trip_id = child.superseded_by` join of the physical-fusion phase (P2) without weighing on the row count of the roots. The `algo_version` column (currently `'v_final_v8'` everywhere) anticipates the side-by-side coexistence of algorithmic versions in a future shadow run.

rt.stib_traveller_message: STIB editorial alerts. The persistence table for STIB editorial alerts ingested by `traveller_info_ingestor.py` (Section 7.4.4). The `content_hash UNIQUE` constraint absorbs republications with identical content, and the four indices accelerate respectively the API route that serves only active messages, the priority pill filter on the frontend, and lookups by impacted line or stop.

```
CREATE TABLE rt.stib_traveller_message (
    message_id    UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    content_hash  TEXT NOT NULL UNIQUE,
    text_fr       TEXT,
    text_nl       TEXT,
    text_en       TEXT,
    priority      SMALLINT,
    line_ids      TEXT[] NOT NULL DEFAULT '{}',
    point_ids     TEXT[] NOT NULL DEFAULT '{}',
    first_seen_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    last_seen_at  TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    resolved_at   TIMESTAMPTZ,
    raw           JSONB NOT NULL DEFAULT '{}'
);
CREATE INDEX idx_stib_tinfo_active
    ON rt.stib_traveller_message(last_seen_at) WHERE resolved_at IS NULL;
CREATE INDEX idx_stib_tinfo_priority
    ON rt.stib_traveller_message(priority);
CREATE INDEX idx_stib_tinfo_lines
    ON rt.stib_traveller_message USING GIN(line_ids);
```

```
CREATE INDEX idx_stib_tinfo_points
    ON rt.stib_traveller_message USING GIN(point_ids);
```

rt.delijn_trip, reconstructed De Lijn trips. The equivalent MobilityDB-backed table for De Lijn trips (Section 7.6). Because the De Lijn GTFS-Realtime feed does carry a stable `vehicle_id`, the matching problem is trivial and the attribution is materialised at insert time.

```
CREATE TABLE IF NOT EXISTS rt.delijn_trip (
    trip_id       TEXT PRIMARY KEY,
    vehicle_id    TEXT NOT NULL,
    gtfs_trip_id  TEXT,
    route_id      TEXT,
    direction_id  INTEGER,
    start_ts      TIMESTAMPTZ NOT NULL,
    end_ts        TIMESTAMPTZ NOT NULL,
    point_count   INTEGER NOT NULL DEFAULT 0,
    trip          tgeompoint,
    ingested_at   TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
CREATE INDEX IF NOT EXISTS idx_delijn_trip_vehicle   ON rt.delijn_trip(vehicle_id);
CREATE INDEX IF NOT EXISTS idx_delijn_trip_gtfs_trip ON rt.delijn_trip(gtfs_trip_id);
CREATE INDEX IF NOT EXISTS idx_delijn_trip_route     ON rt.delijn_trip(route_id);
CREATE INDEX IF NOT EXISTS idx_delijn_trip_start     ON rt.delijn_trip(start_ts);
CREATE INDEX IF NOT EXISTS idx_delijn_trip_gist      ON rt.delijn_trip USING GIST(trip);
CREATE INDEX IF NOT EXISTS idx_delijn_trip_end       ON rt.delijn_trip(end_ts);
```

rt.delijn_traveller_message: De Lijn GTFS-RT Alert entities. The persistence table for De Lijn editorial alerts ingested in piggyback by `delijn_ingestor.py` (Section 7.6.1). The structure is wider than its STIB counterpart to absorb the `cause` and `effect` GTFS-RT enums (stored as `SMALLINT`, with the protobuf integer values), the derived `category` field, the quadrilingual text fields, the serialised representation of multiple active windows in JSONB, and the raw payload preserved for audit.

```
CREATE TABLE rt.delijn_traveller_message (
    message_id     TEXT PRIMARY KEY,
    content_hash   TEXT,
    cause          SMALLINT,
    effect         SMALLINT,
    category       TEXT,
    header_fr      TEXT,
    header_nl      TEXT,
    header_en      TEXT,
    header_de      TEXT,
    text_fr        TEXT,
    text_nl        TEXT,
    text_en        TEXT,
    text_de        TEXT,
    route_ids      TEXT[] NOT NULL DEFAULT '{}',
```

```
    stop_ids       TEXT[] NOT NULL DEFAULT '{}',
    active_start   TIMESTAMPTZ,
    active_end     TIMESTAMPTZ,
    active_periods JSONB NOT NULL DEFAULT '[]',
    first_seen_at  TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    last_seen_at   TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    resolved_at    TIMESTAMPTZ,
    raw            JSONB NOT NULL DEFAULT '{}'
);
CREATE INDEX idx_delijn_tinfo_active
    ON rt.delijn_traveller_message(last_seen_at) WHERE resolved_at IS NULL;
CREATE INDEX idx_delijn_tinfo_cause
    ON rt.delijn_traveller_message(cause);
CREATE INDEX idx_delijn_tinfo_effect
    ON rt.delijn_traveller_message(effect);
CREATE INDEX idx_delijn_tinfo_routes
    ON rt.delijn_traveller_message USING GIN(route_ids);
CREATE INDEX idx_delijn_tinfo_stops
    ON rt.delijn_traveller_message USING GIN(stop_ids);
CREATE INDEX idx_delijn_tinfo_active_p
    ON rt.delijn_traveller_message(active_start, active_end);
```

Unlike the STIB table, the `content_hash` carries no uniqueness constraint: a same `entity.id` can see its content vary over time (text update, period extension), and each variation feeds the lifecycle triple without spawning a new row. The six indices reflect the richness of the frontend filtering criteria.

rt.stib_alert, operational alerts (generic). A single alert table per operator stores every alert type (bunching and the optional `slow_speed` / `stuck` sub-detectors). The `details` JSONB column carries type-specific fields (distances, thresholds, time-band). `rt.delijn_alert` has the same structure except that it keys on `route_id + direction_id + vehicle_id_a/b` rather than STIB-specific fields.

```
CREATE TABLE IF NOT EXISTS rt.stib_alert (
    alert_id        TEXT PRIMARY KEY,
    alert_type      TEXT NOT NULL,
    severity        TEXT NOT NULL DEFAULT 'warning',
    lineid          TEXT NOT NULL,
    direction       INTEGER,
    geom            GEOMETRY(Point, 4326),
    detected_at     TIMESTAMPTZ NOT NULL,
    resolved_at     TIMESTAMPTZ,
    last_updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    details         JSONB NOT NULL DEFAULT '{}',
    trip_id_a       TEXT,
    trip_id_b       TEXT,
    line_trip_id_a  TEXT,
    line_trip_id_b  TEXT,
```

```
    fid             INTEGER,
    created_at      TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
CREATE INDEX IF NOT EXISTS idx_stib_alert_lineid   ON rt.stib_alert(lineid);
CREATE INDEX IF NOT EXISTS idx_stib_alert_detected ON rt.stib_alert(detected_at);
CREATE INDEX IF NOT EXISTS idx_stib_alert_active   ON rt.stib_alert(resolved_at)
    WHERE resolved_at IS NULL;
CREATE INDEX IF NOT EXISTS idx_stib_alert_geom     ON rt.stib_alert USING GIST(geom);
CREATE INDEX IF NOT EXISTS idx_stib_alert_severity ON rt.stib_alert(severity, detected_at);
```

static.stib_line_thresholds, derived view. A computed view, `static.stib_line_thresholds`, derives the high/medium/low alert thresholds per (line, direction, time-band) from `static.stib_line_frequency`. It is refreshed when the static frequency table is reloaded.

Appendix B

Reference MobilityDB Queries

This appendix lists the reference queries used by the RTDataHub API (Section 7.12) and by the benchmark of Section 7.7.4. They illustrate the MobilityDB operators of Section 2.3 in context.

Q1: All positions on line 71 over a 5-minute window.

```
-- Flat equivalent (stib_vehicle_position)
SELECT vehicle_uuid, fetched_at, geom
FROM   stib_vehicle_position
WHERE  lineid = '71'
  AND  fetched_at BETWEEN $1 AND $2;

-- MobilityDB equivalent (stib_trip)
SELECT trip_id, atPeriod(trip, tstzrange($1, $2)) AS trip_slice
FROM   stib_trip
WHERE  lineid = '71'
  AND  start_ts <= $2 AND end_ts >= $1;
```

Q2, Position of a specific vehicle at an exact instant.

```
-- Flat equivalent: linear interpolation between surrounding rows
WITH bracket AS (
  SELECT * FROM stib_vehicle_position
  WHERE  vehicle_uuid = $1 AND fetched_at <= $2
  ORDER BY fetched_at DESC LIMIT 1
), next_row AS (
  SELECT * FROM stib_vehicle_position
  WHERE  vehicle_uuid = $1 AND fetched_at >= $2
  ORDER BY fetched_at ASC LIMIT 1
)
SELECT -- manual interpolation between bracket and next_row
       ...
FROM bracket, next_row;

-- MobilityDB equivalent: single operator call
```

```
SELECT valueAtTimestamp(trip, $2)
FROM   stib_trip
WHERE  trip_id = (SELECT trip_id FROM stib_trip
                  WHERE vehicle_uuid = $1
                    AND $2 BETWEEN start_ts AND end_ts);
```

Q3: Full trajectory of a trip, for animation playback.

```
-- Flat equivalent: order-by-timestamp scan, client-side interpolation
SELECT fetched_at, geom
FROM   stib_vehicle_position
WHERE  trip_id = $1
ORDER BY fetched_at;

-- MobilityDB equivalent: one row, vectorised on the client
SELECT trip FROM stib_trip WHERE trip_id = $1;
```

Q4, Minimum distance between two vehicles over a 10-minute window.

```
-- Flat equivalent: practically infeasible without a temporal join
-- (requires pairwise interpolation between all observations)

-- MobilityDB equivalent: clip both trajectories to the window first,
-- then compute the minimum distance over the clipped pair.
SELECT minDistance(
         atPeriod(a.trip, tstzrange($3, $4)),
         atPeriod(b.trip, tstzrange($3, $4))
       )
FROM   stib_trip a, stib_trip b
WHERE  a.trip_id = $1 AND b.trip_id = $2
  AND  tstzrange($3, $4) && period(a.trip)
  AND  tstzrange($3, $4) && period(b.trip);
```

The `atPeriod` clip ensures the operator returns the minimum distance over the requested 10-minute window, not over the full overlap of the two trips' periods. The outer `&&` predicates remain useful as an early bounding-box filter on the spatio-temporal index before the clip is materialised.

Q5, Bunching detector distance computation. See the code listing in Section 7.11. The detector uses a Haversine pre-filter followed by an along-line distance computation; `minDistance` is a conceptual baseline against which the two-pass custom metric is compared in Section 7.7.4.

Appendix C

Bunching Detector Configuration

This appendix reproduces the full configuration of the automated bus-bunching detector described in Section 7.11, for readers who want to reproduce or adapt the detector. Unlike the other RTDataHub modules, the bunching detector is not configured through a YAML file: its parameters are Python constants defined at the top of `src/etl/pipeline/load/stib_alert_detector.py` and `src/etl/pipeline/load/delijn_alert_detector.py`. Most parameters are shared between the two detectors; the freshness gate `PAIR_MAX_STALENESS_S` differs per feed, for the reasons discussed in Section 7.11.4. The snippet below is a verbatim extract of the STIB constants; the De Lijn block follows.

```
# ---- Distance thresholds (metres) --------------------------------
FALLBACK_THRESHOLDS   = {"high":   30.0,
                         "medium": 80.0,
                         "low":    200.0}
SEVERITY_FRACTIONS    = {"high":   0.05,   # 5 % of planned headway-distance
                         "medium": 0.12,   # 12 %
                         "low":    0.20}   # 20 %
MIN_THRESHOLD_M       = 20.0
MAX_THRESHOLD_M       = 500.0
HAVERSINE_PREFILTER_M = 600.0  # skip pairs further than this before MobilityDB
MIN_PAIR_DISTANCE_M   = 10.0   # reject noise below this floor

# ---- Freshness gate on paired vehicles (STIB) --------------------
PAIR_MAX_STALENESS_S  = 180.0  # see rationale below; calibrated on the
                               # post-enrichment empirical distribution,
                               # observed median 20s, p95 81s, p99 282s.

# ---- Stopped-vehicle suppression ---------------------------------
STOPPED_THRESHOLD_S       = 180   # 3 minutes
STOPPED_MIN_DIST_STOP_M   = 80    # within 80 m of a stop => treat as layover

# ---- Time-of-day bands (local time) ------------------------------
TIME_BANDS = [("morning_peak", 7, 9),
              ("daytime",      9, 16),
              ("evening_peak", 16, 19),
              ("evening",      19, 22),
```

```
                ("night",        22, 7)]   # night wraps midnight
```

The De Lijn detector reuses every distance threshold, severity fraction, clamp, stopped-vehicle suppression and time-band of the STIB block above. The single per-feed difference is the freshness gate, set to 60 s rather than 180 s:

```
# ---- Freshness gate on paired vehicles (De Lijn) ----------------
PAIR_MAX_STALENESS_S  = 60.0   # three native polling cycles (3 x 20 s);
                               # tighter than STIB because the De Lijn
                               # feed delivers a direct GPS reading with
                               # a stable vehicle_id, with no intermediate
                               # odometric-projection step.
```

Parameter rationale. The detection cadence of one pass per minute is set at the scheduler level (Section 7.3) and matches the one recommended by Cats [52] for operationally useful alerts. The 600 m Haversine pre-filter is conservative and retains a generous margin over the largest plausible along-line gap the threshold rule might accept. The `SEVERITY_FRACTIONS` values (5/12/20 % of the planned headway-distance) come from the classical bunching literature [17] adapted to this frequency-aware formulation; the `FALLBACK_THRESHOLDS` (30/80/200 m) are the hard-coded floors used when no line-specific or global threshold is available in `static.stib_line_thresholds`. The two `MIN_/MAX_THRESHOLD_M` clamps (20 m and 500 m) prevent pathological low-frequency lines from producing either noise-level or city-scale alert radii.

The `PAIR_MAX_STALENESS_S` freshness gate is the safeguard introduced in Section 7.11.4: it rejects any pair whose either vehicle's `last_seen` is older than the per-feed threshold relative to the batch reference timestamp, eliminating phantom alerts between live vehicles and frozen last-known positions of trips that finished inside the 30-minute `TRIP_TIMEOUT_S` grace window. The threshold differs per feed because the two pipelines feed the detector with `last_seen` distributions that differ in shape. On STIB, the position is the output of the proximity matcher (Section 7.4.1), which projects an odometric position onto a reconstructed trajectory; ODP-side polling misses compound with the enrichment delay of this projection, so the empirical `last_seen` distribution has a long tail (median 20 s, p_{95} ≈81 s, p_{99} ≈282 s). The 180 s threshold sits between p_{95} and p_{99}, admitting most legitimate enrichment lag while rejecting outright-stale positions. On De Lijn, the feed delivers a direct GPS reading with a stable `vehicle_id`, with no intermediate odometric-projection step, so the intrinsic staleness is dominated by upstream polling alone. The 60 s threshold matches three native polling cycles (3×20 s) and yields a strictly tighter gate.

The `STOPPED_*` pair suppresses alerts when both vehicles have been stationary near a stop for more than three minutes, which absorbs most terminal-layover false positives (a cleaner fix using `stops.txt` annotations is on the future-work agenda, Section 11).

Day typing. The Belgian-calendar context used by the threshold lookup (five day types: weekday, Saturday, Sunday, public holiday, school vacation) is *not* in this module. It is computed once per day by a separate calendar helper that combines Python's `datetime.weekday()`, a static public-holiday table, and Meeus's Gregorian Easter algorithm [69] for the movable holidays (Easter Monday, Ascension, Pentecost Monday).

Appendix D

Lines Covered by RTDataHub

This appendix lists the STIB and De Lijn lines ingested by RTDataHub during the 24-hour (STIB) and three-day (De Lijn) captures used to produce the numerical results of Section 7. The list is dynamic: it is regenerated every time the ingestion layer picks up a new GTFS static snapshot (Section 7.3); the tables below reflect the snapshot in effect at the end of the 2026-04-23 measurement window.

STIB-MIVB coverage. The STIB-MIVB ingestion module captures every vehicle reporting on the operator's stateless position feed [7]. Modes are derived by joining `rt.stib_vehicle_position` against `static.stib_line_shape`. The following 73 lines were active during the 24-hour window: 4 metro + 18 tram + 51 bus.

```
Metro:   1, 2, 5, 6
Tram:    4, 7, 8, 9, 10, 18, 19, 25, 35, 39, 44, 51, 55, 62,
         81, 82, 92, 93
Bus:     12, 13, 14, 17, 20, 21, 28, 29, 34, 36, 37, 38, 41,
         42, 43, 45, 46, 47, 48, 49, 50, 52, 53, 54, 56, 58,
         59, 60, 61, 63, 64, 65, 66, 69, 71, 72, 73, 74, 75,
         76, 77, 78, 79, 80, 83, 86, 87, 88, 89, 95, 96
```

De Lijn coverage. The De Lijn ingestion module captures every vehicle reporting on the operator's GTFS-Realtime `VehiclePosition` feed [8]. The selection is filtered to vehicles whose position falls within the Brussels-Capital bounding box `(4.24, 50.76, 4.49, 50.92)` in EPSG:4326. Over the three-day window, 76 distinct `route_id` values intersected that bounding box. De Lijn publishes internal `route_id` codes rather than the passenger-facing short names; mapping to short names requires a refresh of the GTFS `routes.txt` static bootstrap (Section 7.3), which was not refreshed on this measurement window.

```
Brussels-bound routes (internal route_id):
  3003, 3010, 3070, 3071, 3075, 3076, 3077, 3078, 3079, 3080,
  3081, 3090, 3091, 3092, 3110, 3115, 3116, 3118, 3124, 3126,
  3128, 3129, 3136, 3142, 3152, 3153, 3154, 3155, 3156, 3165,
  3170, 3171, 3172, 3191, 3201, 3212, 3213, 3214, 3216, 3220,
  3221, 3225, 3226, 3227, 3228, 3230, 3231, 3240, 3241, 3245,
  3246, 3250, 3259, 3260, 3282, 3460, 3470, 3504, 3505, 3530,
  3536, 3538, 3543, 3546, 3547, 3553, 3554, 3555, 3570, 3571,
  3572, 3573, 3595, 3598, 3620, 3819
```

Aggregate statistics. Read off the `rt.*` tables at an intermediate snapshot of the 2026-04-23 window (taken when the lines-coverage analysis was run, several hours before the end-of-day cutoff):

- STIB (24-hour window, intermediate snapshot): **3 617** distinct trips (by `line_trip_id`), **1 625 627** raw position observations, **61 655** reconstructed `tgeompoint` rows in `rt.stib_trip`.
- De Lijn (three-day window, Brussels-boundary filter applied): **7 210** distinct trips (by `gtfs_trip_id`), **1 373 563** raw observations after filtering.

The figures in Table 7.10 of Section 7.7.4 (1 835 566 raw STIB rows, 68 114 `stib_trip` rows for the same date) come from a later end-of-day probe of the same tables; the appendix figures are reported separately because they were the inputs to the lines-coverage CSV dumps preserved in the companion repository, so that the numbers in Section 7.7.4 can be reproduced against exactly the same data.

Appendix E

Algorithm Evolution: `greedy_haversine` → `greedy_chainmerge`

This appendix carries the per-version detail behind the seven-day benchmark of Section 7.5: the full per-version metric table, a paragraph of commentary on each step of the design lineage, the variants that were tested and dropped, and the forensic diagnostic that motivated the feed-dropout merge. Seven matchers traverse the design space, from a Haversine-only greedy assignment (`greedy_haversine`) to the two-stage matcher deployed in production (`greedy_chainmerge`); the vendored MobilityTwin algorithm [65] is included as the external baseline. The lineage trajectory is plotted in the body (Figure 7.3, Section 7.5.3).

E.1 Reference Dataset and Methodology

All numbers in this appendix come from the seven-day benchmark of Section 7.5: a continuous capture of the live STIB feed from 2026-05-04 to 2026-05-10, with the GTFS static feed overlapping the window (78 921 scheduled courses) as the schedule-derived reference. Each matcher is run from the same input loader, with the same schedule-anchoring harness; only the matcher itself differs between rows. The metric definitions are those of Section 7.5.2.

E.2 Per-Version Metric Table

Table E.1 reports the per-version metrics on every retained matcher, plus the GTFS schedule and the vendored MobilityTwin baseline.

E.3 Per-Version Commentary

`greedy_haversine` (the baseline matcher). The starting point: a greedy one-to-one assignment on Haversine distance between a trip's last observation and the new observation, with the static 500 m / 300 s gates of the v1 loader. With no along-shape progression test, two physical vehicles heading the same way within 500 m of each other are routinely swapped between trips, and the matcher emits short fragments instead of consistent long trips. The seven-day capture shows the structural cost: a 25.41% fragmentation rate, 495 419 reconstructed trips against 78 921 scheduled courses, and a 6.7% median stop coverage. The row is reported as the naive-baseline control.

Table E.1: Per-version metrics on the seven-day STIB capture. count_score is ideal at 1; for the rest, ↓ = lower is better, ↑ = higher is better. anch_pct is a diagnostic, not a quality metric: it is inflated on fragmented matchers (`greedy_haversine`, MobilityTwin) by false-positive anchoring, where short fragments accidentally satisfy the temporal-overlap criterion of the anchoring step (Section 7.5.2). Values are taken from the seven-day benchmark `compare.csv`.

Algorithm	n_trips	count→ 1	frag %↓	hi_sp %↓	anch %	med_cov %↑	clean %↑
`greedy_haversine`	495 419	8.01	25.41	0.263	93.0	6.7	20.8
`curvilinear`	165 571	2.30	2.27	0.082	68.3	42.9	22.0
`hungarian`	164 192	2.24	2.01	**0.074**	62.9	42.4	22.1
`hungarian_snap`	167 792	2.61	1.76	0.079	62.9	42.9	19.5
`hungarian_terminus`	161 415	2.46	1.80	0.077	60.5	42.9	21.0
`hungarian_merge`	156 520	2.32	1.69	0.120	59.5	44.4	22.8
`greedy_chainmerge`	**97 807**	**1.34**	**0.49**	0.333	40.2	**54.5**	**25.4**
MobilityTwin	342 709	5.77	11.64	2.149	89.8	21.4	13.9
GTFS (ref.)	78 921	1.00	0.14	0.000	100.0	100.0	100.0

`curvilinear` (the along-shape progression gate). Replaces the Haversine gate with the per-line curvilinear gate of Section 7.4.1, which enforces plausible progress *along the route shape* rather than mere straight-line proximity. This is the largest single step of the trajectory: fragmentation falls from 25.41% to 2.27%, the trip count from 495 419 to 165 571, and the median stop coverage rises from 6.7% to 42.9%. On shared trunks where several routes overlap geometrically, the curvilinear projection is taken against each vehicle's own declared route shape; the residual ambiguity is bounded by the `(line, direction)` key but not fully eliminated.

`hungarian` (Hungarian assignment per batch). Switches the per-batch assignment from greedy arg min to the Hungarian solver [63] on the bipartite (observations, open trips) cost matrix, with virtual "new trip" columns. The headline metrics move only marginally (fragmentation 2.27 → 2.01%) because the structural gain was already delivered by the curvilinear gate. Hungarian is retained for its behaviour on the rare dense batches, where a greedy assignment makes locally-optimal swaps the next batch cannot undo; the impossible-speed rate reaches its lineage minimum here (0.074%).

`hungarian_snap` (dynamic windows, duration cap, stop-snap). Three changes ship together: the temporal match window is computed per line from the observed cadence rather than fixed; a per-line trip-duration cap stops the matcher extending one trip across a layover; and the interpolator snaps to a scheduled stop within a tolerance of the observation timestamp. Fragmentation continues down to 1.76%, though the clean-trip share dips to 19.5% as the duration cap surfaces short-turn trips previously agglomerated into longer fragments.

`hungarian_terminus` (terminus-aware splitting). The duration cap is now consulted only after a terminus check: enforced as a hard reject when the trip ends in the line's terminus zone (a legitimate turnaround), relaxed otherwise (a mid-route delay). This recovers the clean-trip share (19.5 → 21.0%) without losing ground on fragmentation (1.76 → 1.80%). By construction this version is the input to the feed-dropout merge.

`hungarian_merge` (zero-risk anchor-aware feed-dropout merge). Adds the post-process of Section 7.4.1: contiguous fragments that plausibly belong to one trajectory are fused, unless the

schedule-anchoring step assigns the two halves to distinct scheduled trips. Fragmentation reaches 1.69%, the minimum of the Hungarian family; median coverage rises to 44.4% and count_score tightens to 2.32. This is the design endpoint of the offline trajectory, and the matcher admitted to first production deployment.

`greedy_chainmerge` (two-stage streaming matcher). The operational specialisation deployed in continuous production, described in Section 7.4.3: a streaming Stage A and an hourly chain-merge replace the anchor-aware merge. On the seven-day capture it leaves the Hungarian plateau on every quality axis: fragmentation 0.49%, median coverage 54.5%, clean-trip share 25.4%, and count_score 1.34, the closest of the lineage to the ideal of 1. The measured cost is a rise in the impossible-speed rate to 0.333%, analysed in Section 7.5.5.

E.4 Trip Duration and GPS-Point Density

Figure E.1 characterises the reconstructed trips of each matcher by two distributions: trip duration and the number of GPS observations per trip. The two move together along the lineage. `greedy_haversine` produces short, sparse fragments, a median of six observations per trip; `greedy_chainmerge` produces the longest and densest trips, a median of sixty-two observations, the signature of a matcher that assembles each service into one coherent trajectory rather than scattering it across fragments. MobilityTwin sits closer to the fragmented end, at a median of thirteen observations per trip.

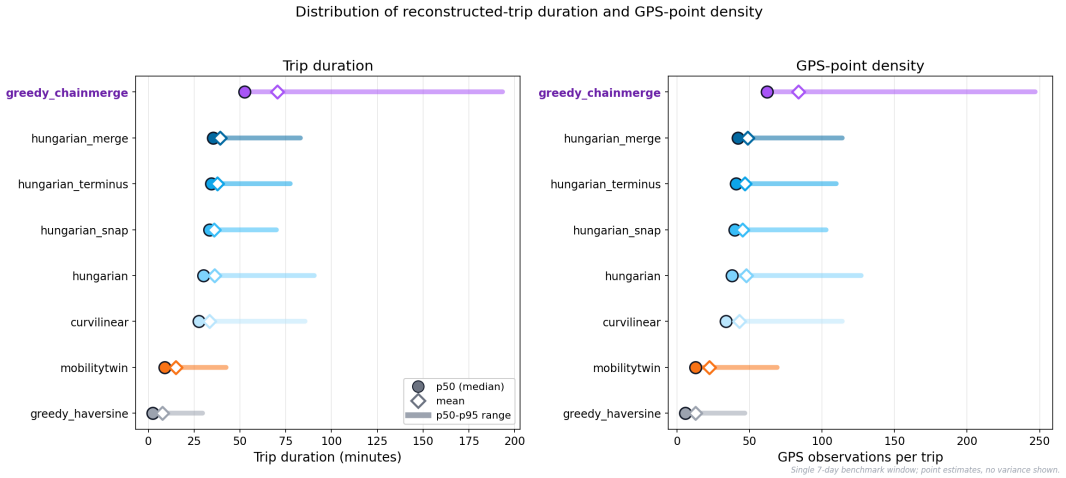


Figure E.1: Distribution of reconstructed-trip duration (left) and GPS observations per trip (right), per matcher, on the seven-day capture. The two move together along the lineage: `greedy_chainmerge` produces the longest and densest trips, `greedy_haversine` the shortest and sparsest. Single seven-day benchmark window.

E.5 Versions That Were Tested and Dropped

Three additional variants were benchmarked and rejected. Table E.2 summarises their failure modes; their collective lesson shaped the design of the feed-dropout merge.

Table E.2: Variants benchmarked and dropped, with the failure that caused the rejection.

Variant	Idea	Why it was dropped
`hungarian_terminus-A`	Bridge feed dropouts inside the Hungarian matcher by widening the temporal window when curvilinear evidence supported continuation.	The matcher cannot distinguish "same vehicle resumes" from "next scheduled trip starts here". The dropout decision belongs in a post-process with anchoring evidence, not inside the matcher.
`hungarian_terminus-PID-K3`	Extend the curvilinear validity check with a stop-approach prior.	Valid mid-segment continuations were spuriously rejected, raising the fragmentation rate.
`hungarian_merge GEO_ONLY`	Restrict the merge to the GEOMETRIC pattern, dropping the SAME-PID pattern.	Too few merges fired for any measurable metric movement; the SAME-PID pattern carries the bulk of the safe merges.

E.6 feed_dropout Diagnostic — Additional Detail

The forensic diagnostic that motivated the feed-dropout merge is reproduced and discussed in Section 7.4.1 of the body, where the classification table of `diag_v12_non_clean.py` is shown alongside the description of the merge it motivated. The operational definition is repeated here for completeness: a trip t is classified FEED_DROPOUT when, in the same (*line*, *direction*) group, there exists another trip whose start lies in $(t.end, t.end + 600\,\text{s}]$ and within 2 km of $t.end$, or symmetrically a predecessor in the matching window before $t.start$. The merge fuses only the pairs that satisfy at least one of the two zero-risk patterns of Algorithm 7.2 and the anchor-aware guard, that is, the subset of dropout candidates that can be fused without compromising anchoring quality.

E.7 Reproducibility

The benchmark is fully scripted: the figures and tables of Section 7.5 and of this appendix are regenerated from a single `compare.csv`, itself produced by replaying a captured STIB feed through each matcher in turn. The vendored MobilityTwin algorithm is preserved in `src/comparison/mobilitytwin/vendored/` together with its upstream `LICENSE_AITwin`.

Appendix F

Cross-Architecture Benchmark: Per-Condition Technical Notes and Reproducibility

This appendix supports Chapter 6 with the technical material that would have interrupted the narrative if placed inline: a more detailed description of each of the six conditions, the dataset profiles, the per-cell measurements as a single table, and the reproducibility pointers (hardware, software versions, commands, raw output).

F.1 Per-Condition Technical Description

Each of the six conditions of Chapter 6 corresponds to a self-contained pipeline. The descriptions below summarise what each pipeline does between the moment the user opens an animated MobilityDB layer in QGIS and the moment a frame appears on the canvas.

C1 — Ali naïve (the published 2024 baseline). The pipeline of Manzer [2]. All trips are loaded eagerly into a Python dictionary indexed by trip identifier. A QGIS memory layer is created with one feature per trip; the geometry is left empty initially. On every frame timestamp t produced by the Temporal Controller, the plugin iterates over all N trips, calls `value_at_timestamp(t)` on each, converts the result to a Shapely `Point` [74], then to WKB, then to a `QgsGeometry`, and finally pushes the geometry into the memory layer via per-feature `setGeometry` followed by a final `commitChanges`. The exception handling is broad: a `try/except Exception` surrounds the per-trip interpolation, which under stress (most trips inactive at t) makes the exception path the dominant cost. Frame timing is taken around the `commitChanges` call; canvas paint is not measured.

C2 — Ali optim (targeted optimisation of C1). The C1 architecture is preserved end-to-end, but three Python hot spots identified by cProfile and `viztracer` profiling have been removed. *(i) Temporal pre-filter.* A SQL view exposes `(trip_id, t_start, t_end)` per trip; the plugin queries it once at start-up into an in-memory array and uses it to skip inactive trips before the `value_at_timestamp` call. The broad `try/except` is replaced by a narrow `except TypeError` that catches only the rare PyMEOS gap path. *(ii) Vectorised WKB construction.* The per-feature `Shapely.Point` → WKB conversion is replaced by a NumPy broadcast over the 21-byte fixed-layout WKB of a 2D point: one buffer allocation per frame, no per-trip Python

object. *(iii) Bulk `changeGeometryValues`.* The per-feature `setGeometry` loop is replaced by a single `changeGeometryValues(fid: geom)` call that batches all geometry updates of the frame. The architecture itself (per-trip Python loop, in-memory state dictionary, memory layer, no expression engine) is unchanged; only the Python-side hot path has been optimised. The headline lift from ∼17 FPS to ∼181 FPS on STIB at $N = 5\,000$ is attributable to these three targeted changes.

C3 — MOVE Fast Preview. The `MoveTrajectoryItem` subclass of `QgsMapCanvasItem` bypasses the QGIS feature provider entirely and draws onto the canvas via `QPainter` directly. Per-trip geometry is held in a pre-computed array indexed by frame; the per-frame `paint(QPainter)` method iterates over the array and emits the corresponding drawing calls. Because the canvas-item path replaces both the provider and the renderer, the C3 frame timing covers *compute + paint* as measured at the `QPainter.end()` call. The pipeline preserves no QGIS layer abstraction; downstream consumers (identify tool, attribute table, snapping) are unavailable.

C4 — Columnar (server-side pre-sampling + NumPy matrix). At load time, a single SQL query applies MobilityDB's `tsample()` to every trip on a regular temporal grid, aggregates the resulting per-trip coordinate arrays server-side, and ships them back in a single round-trip. Two NumPy matrices $\mathbf{X}, \mathbf{Y}$ of size $N \times T$ are constructed in float64; inactive cells are `NaN`. The per-frame cost is then $\mathbf{X}[:, t] + \mathbf{Y}[:, t]$ + a vectorised NaN mask + the same vectorised WKB construction as C2 + a bulk `changeGeometryValues`. Memory cost is the dominant deployability concern: $N \times T \times 2 \times 8$ bytes for the matrices, plus PyMEOS load-time buffers that are released after the matrix is assembled. The pre-sampling step takes ∼8 s on AIS at $N = 5\,000$ and ∼54 s at $N = 17\,118$. Frame timing covers compute-only; canvas paint is not measured.

C5 — MOVE upstream (current main of the plugin). The current main branch of the MOVE plugin [62]. A QGIS layer backed by the PostgreSQL provider is connected to the live MobilityDB table. The trajectory animation is driven by a QGIS expression on the layer, `geometry_n(..., line_interpolate_point(..., t))`: on every frame, the expression engine iterates over all visible features, evaluates the expression once per feature, and produces the per-feature geometry that the renderer paints. The provider call (PostgreSQL round-trip) and the expression evaluation are both per-frame costs; the canvas paint uses `QgsMapRendererSequentialJob`. Frame timing covers *compute + raster render* measured at the `renderingFinished` signal.

C6 — MOVE upgrade (candidate upstream patch). A small, opt-in patch to the MOVE plugin [62]: when the user enables the "memory layer" flag, the plugin replaces the PostgreSQL provider with an in-process memory-layer provider, pre-populated once at start-up from the same MobilityDB queries that the upstream issues every frame. Everything else of the pipeline is unchanged: the same QGIS expression engine, the same `line_interpolate_point` call, the same raster renderer. The patch is therefore a provider swap, not an expression-engine bypass. At the time of writing, the patch is in discussion with the plugin maintainer (Maxime Schoemans, ULB) and has not been merged into the main branch. Frame timing covers *compute + raster render* as for C5.

F.2 Dataset Profile

The two datasets are identical to those used by Manzer [2], subset to $N = 5\,000$ trips each so that the six conditions are run on identical cardinalities. Table F.1 summarises the geometric profile of the two subsets, which is the load-bearing parameter for the slope analysis of Section 6.6.

Table F.1: Geometric profile of the two datasets used in the cross-architecture benchmark, subset to $N = 5\,000$ trips each.

Dataset	Trips (N)	Mean vertices/trip	Sampling cadence
STIB Brussels	5 000	≈ 14	∼30 s vehicle-position poll
Danish AIS	5 000	$\approx 1\,230$	∼60 s position broadcast

F.3 Per-Cell Measurements (Full Table)

Table F.2 reproduces every cell of the 6×2 matrix as a single table. The CI columns report observed extremes over four per-run medians at $n = 4$ effective (cf. Section 6.2), not statistically rigorous 95% confidence intervals. The source data file is `bench5_cross_matrix.csv` in the companion repository.

Table F.2: Full per-cell measurements of the cross-architecture benchmark. Latency reported as median over the four retained runs; observed extremes in brackets. CPU% is the mean process CPU utilisation during the run; RAM is the peak RSS delta over the pre-run baseline of the same QGIS session.

Condition	Dataset	Median (ms)	Observed range (ms)	FPS	RAM Δ (MB)
C1 Ali naïve	STIB	60.5	[58.3, 62.9]	16.5	+26.9
C1 Ali naïve	AIS	127.9	[126.9, 129.7]	7.8	+455.2
C2 Ali optim	STIB	5.5	[5.5, 5.7]	181.1	+25.6
C2 Ali optim	AIS	15.4	[15.1, 15.7]	64.9	+512.5
C3 MOVE Fast Preview	STIB	13.3	[13.0, 13.6]	75.3	+16.6
C3 MOVE Fast Preview	AIS	67.0	[66.4, 67.8]	14.9	+712.8
C4 Columnar	STIB	22.9	[22.0, 23.6]	43.7	+892.8
C4 Columnar	AIS	28.6	[27.1, 29.3]	35.0	+759.4
C5 MOVE upstream	STIB	120.4	[118.2, 133.6]	8.3	+1.2
C5 MOVE upstream	AIS	550.3	[537.5, 565.9]	1.8	+363.5
C6 MOVE upgrade	STIB	7.5	[7.3, 7.8]	133.7	+0.1
C6 MOVE upgrade	AIS	126.5	[125.8, 127.4]	7.9	+0.0

F.4 Hardware and Software Versions

All measurements were taken on the benchmark VM described in Section 5.1: Ubuntu 25.04 guest, 4 vCPU, 7.2 GiB RAM, no GPU, host-backed SSD virtual disk. Software pinning at the time of the final measurements:

- QGIS 3.42.0 Münster (the LTR 3.34 is not tested in this benchmark);
- PostgreSQL 18.0 + PostGIS 3.6.0 + MobilityDB 1.3.0, two local instances (one per dataset, ports 5432 and 5433);
- Python 3.12.12;
- PyMEOS 1.3 [23];

- the MOVE plugin [62] `main` branch at the time of the benchmark (used in C5 unmodified; the candidate C6 patch is a local branch);
- Operating-system kernel Linux 6.14.0-37-generic.

F.5 Reproducibility

The benchmark chain is fully scripted. Inside a fresh QGIS Python console:

```
import sys, os
os.environ['BENCH_DATASET'] = 'stib'         # or 'ais'
os.environ['BENCH_LIMIT']    = '5000'
os.environ['BENCH_N_TRIPS']  = '5000'
os.environ['BENCH_N_FRAMES'] = '60'
os.environ['BENCH_N_RUNS']   = '5'
exec(open('bench5_master.py').read())
```

The two datasets are run in successive QGIS sessions to avoid the session-state contamination documented in Refutation Table row 7 (Section 6.8). The aggregation step is then run outside QGIS:

```
python3 bench5_aggregate.py
```

The aggregator produces `bench5_cross_matrix.csv` (the data file behind Table F.2) and a text summary. The four figures of Chapter 6 are regenerated from the CSV by `bench5_charts.py`.

The companion repository contains the scripts (`bench5_c1_ali_naive.py`, `bench5_c2_ali_optim.py`, `bench5_c3_fast.py`, `columnar.py`, `move.py`, `move_inmemory.py`), the aggregator, the chart generator, and the raw per-run JSON dumps from which the CSV is built. Quality-of-life caveats are documented in the repository's `BENCH5_README.md`: in particular the differing default `N_FRAMES` between conditions (C1/C2/C3 read `BENCH_N_FRAMES`, C4/C5/C6 use a fixed $N\_FRAMES\_SYNTHETIC = 120$; medians remain comparable but tail percentiles are computed over different frame counts), and the necessary QGIS restart between datasets.