

The background of the page features a large, light blue watermark of the University of Freiburg seal. The seal is circular and contains a central figure of a seated scholar, surrounded by various heraldic symbols and Latin text.

Bachelor's Thesis

gtfs2rdf: An Efficient, Customisable GTFS-to-RDF Converter

Jan Babin

16 March 2026

University of Freiburg
Department of Computer Science
Chair for Algorithms and Data Structures

universität freiburg

University of Freiburg
Department of Computer Science
Chair for Algorithms and Data Structures

Author Jan Babin,
 Matriculation Number: 5307139

Editing Time 15 December 2025 - 16 March 2026

Examiner Professor Hannah Bast,
 Department of Computer Science
 Chair for Algorithms and Data Structures

Supervisor Dr Patrick Brosi,
 Department of Computer Science
 Chair for Algorithms and Data Structures

Declaration I hereby declare that I am the sole author and composer of this thesis and that no other sources or aids other than those disclosed in this thesis have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I hereby also declare that my thesis has not been prepared for another examination or assignment, neither in its entirety nor in part.

Place, Date

Signature

Declaration on the Use of AI

During the preparation of this thesis, generative AI tools have been utilised to support (i) outlining and structuring text, (ii) improving phrasing and readability, (iii) generating \LaTeX code for figures and tables, (iv) brainstorming and providing feedback on design decisions, and (v) generating code suggestions.

All generated content was carefully reviewed, edited, and adapted by me. I am responsible for the final content of this work, including its correctness and proper attribution.

Specifically, the following tools were used:

- ChatGPT (OpenAI).
- GitHub Copilot (GitHub).

Abstract

We present `gtfs2rdf`, a tool for converting public transport schedule data in GTFS format into RDF triples and thus into a knowledge graph. The tool provides a flexible mapping interface, supports custom data transformations, and offers additional functionality that prepares generated RDF data for more meaningful downstream integration with other datasets and flexible information retrieval. At the same time, `gtfs2rdf` achieves linear-time behaviour for purely row-wise mappings and a configurable memory-usage bound for all mappings, which enables the processing of large real-world feeds on commodity hardware. Experimental evaluation and comparison with related tools suggest that `gtfs2rdf` successfully fills an existing gap in GTFS-to-RDF conversion.

Zusammenfassung

Diese Arbeit präsentiert `gtfs2rdf`, ein Programm zur Konvertierung von Fahrplandaten des öffentlichen Verkehrs im GTFS-Format in RDF-Tripel und damit in einen Wissensgraphen. Das Werkzeug bietet eine flexible Mapping-Schnittstelle, unterstützt benutzerdefinierte Datentransformationen und stellt zusätzliche Funktionen bereit, die die erzeugten RDF-Daten für eine Integration mit anderen Datensätzen sowie flexible Informationsabfragen vorbereiten. Zugleich erreicht `gtfs2rdf` für rein zeilenweise Mappings lineare Laufzeit und gewährleistet für alle Mappings ein konfigurierbares Arbeitsspeicher-Limit, sodass auch große reale Datensätze auf Standardhardware verarbeitet werden können. Die experimentelle Evaluation sowie der Vergleich mit verwandten Konvertierern deuten darauf hin, dass `gtfs2rdf` eine bislang bestehende Lücke bei der Konvertierung von GTFS nach RDF erfolgreich schließt.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Research Questions	2
1.3	Outline	3
2	Background and Related Work	5
2.1	GTFS	5
2.2	RDF	7
2.3	SPARQL and SPARQL Endpoints	9
2.4	Related Converters	11
2.4.1	GTFS-to-RDF Converters	11
2.4.2	Generic CSV-to-RDF Converters	11
2.4.3	RML/KGC Engines	12
2.5	Requirements and Contributions	13
3	System and Implementation	15
3.1	Overview and Design Goals	15
3.1.1	Core Concepts and Terminology	16
3.2	Mapping and DSL	17
3.3	Transforms	19
3.4	Cross-File Persistence	21
3.4.1	Aggregation	22
3.4.2	SQL-backed Storage	23
3.4.3	Topological Sorting	24
3.5	Schema Lifecycle	25
3.5.1	Initialisation	25
3.5.2	Compilation	26
3.5.3	Binding	26

3.6	Streaming Execution: Parsing, Rendering, Writing	27
3.6.1	Per-file execution flow	27
3.6.2	Streaming CSV Parsing	28
3.6.3	Triple Rendering	29
3.6.4	Output Writing	31
3.7	Diagnostics	31
3.8	Testing	32
3.9	Efficiency Considerations	32
4	Analysis and Evaluation	35
4.1	Theoretical Analysis	35
4.1.1	Time Complexity	35
4.1.2	Space Complexity	36
4.2	Experimental Analysis	37
4.2.1	Metrics	37
4.2.2	Setup	37
4.2.3	Data	38
4.2.4	Results	39
5	Conclusion and Future Work	45
5.1	Conclusion	45
5.2	Future Work	46
	Bibliography	51
	Appendix	A
1	Command-line interface	A
2	Mapping DSL Reference	B
2.1	Placeholder basics	B
2.2	Placeholder forms (summary)	C
2.3	Arguments	C
2.4	Transforms and chaining semantics	D
2.5	Storage writes and keyed writes	D
2.6	Contexts and dependency visibility	E
2.7	Storage-only user statements	F
2.8	Canonical examples	F
2.9	Implementation limits	G

3	GTFS Reference Graph	G
---	--------------------------------	---

Chapter 1

Introduction

1.1 Motivation

GTFS (*General Transit Feed Specification*) is a widely used specification for representing public transportation data. It consists of two parts: static schedule information and real-time information. This work is concerned with the former [1].

Owing to its simplicity, GTFS has been adopted by many transit agencies around the world and is being actively maintained by a large community of developers and users, offering an advanced ecosystem of tools and libraries for working with GTFS data. Due to this wide uptake, it may be considered the de facto standard for public transportation data. Today, there are GTFS feeds as large as several gigabytes, and the size of feeds will likely grow even larger in the future, as more transit agencies adopt GTFS and as the amount of data collected and shared by transit agencies increases.

At the same time, GTFS feeds are relational, consisting of multiple CSV files, and highly heterogeneous. While these characteristics are not inherently problematic, they can make efficient data retrieval and the integration of GTFS data with other sources, such as OpenStreetMap or Wikidata, more challenging [2, 3]. Such integration could, however, place GTFS feeds, which are primarily concerned with schedule and transit-network information, into a broader geographic and encyclopedic context, thereby enabling more expressive queries over the combined data. One might think of questions such as the following:

- *Show me the exact path a selected line takes through the city, and highlight where it stops.*

- *Name all stops within 250 metres of a hospital.*
- *Show me all lines that serve a major museum of the city, and what is each museum called in the local language and in English?*

A promising remedy is RDF (*Resource Description Framework*), a graph-based data model that allows for flexible and efficient representation of complex data [4]. Moreover, datasets adhering to RDF standards can be integrated with other RDF datasets with comparatively little effort, and there are many tools and libraries available for working with RDF data. Specifically, RDF data can be queried efficiently using SPARQL, a powerful query language for RDF data [5], and search systems for RDF data, such as *QLever* [6] or Ontotext’s *GraphDB* [7].

Therefore, it would be highly desirable to have a tool that can efficiently convert GTFS data into RDF, so that GTFS can continue to serve as a simple and maintainable source format, while RDF can be generated on demand for efficient querying and seamless integration with other datasets.

1.2 Research Questions

This thesis addresses the problem of converting large and heterogeneous GTFS feeds into RDF in a way that is both efficient and flexible. Concretely, we investigate the following research questions:

1. **(RQ1: Scalability)** How can large, real-world GTFS feeds be converted to RDF efficiently with respect to running time and memory consumption?
2. **(RQ2: Mapping Flexibility)** How can GTFS-to-RDF mappings be expressed such that they remain adaptable to heterogeneous feeds and varying user needs, while accommodating the evolving GTFS specification?
3. **(RQ3: Multi-file Semantics)** How can a converter support semantics that require information across multiple GTFS files while preserving efficient execution?
4. **(RQ4: Comparison)** How does a specialised GTFS-to-RDF converter compare with existing GTFS-specific and generic conversion tools in terms of performance and practical applicability?

1.3 Outline

To achieve a thorough understanding of the problem, the design and implementation of the proposed solution, and its analysis and evaluation, this thesis aims to introduce concepts and methods in a way that is incremental, self-contained and following a natural flow.

In Chapter 2, we will cover the public transport data format GTFS and the knowledge graph RDF data model and explain how converting GTFS to RDF can be beneficial for downstream applications, especially for fast information retrieval and visualisation. By analysing how related conversion tools are still lacking in this area, we will further motivate the need for a specialised GTFS-to-RDF converter and also derive requirements and contributions for the proposed solution. Key findings include the need for a streaming conversion pipeline for performance and scalability, a flexible mapping interface to accommodate heterogeneous feeds and evolving specifications, and the ability to perform data transformations and cross-file lookups to support complex semantics.

Chapter 3 will approach the design and implementation of our tool `gtfs2rdf` by first looking at the user interface and configuration options. These are designed so that a user can directly specify the output pattern, i.e. the mapping from GTFS to RDF, to fit their needs. A simple domain-specific language embedded in C++ enables users to express more complex mappings, including data transformations and cross-file lookups. The latter is enabled by a storage component that offloads storage to disk to avoid unbounded memory consumption and thereby allows usage on large input feeds. The implementation details laid out in the second part of the chapter will furthermore illustrate that as much work as possible is done before the actual conversion loop, so that the conversion itself can be executed as efficiently as possible. This includes the parsing of the mapping, the preparation of static output text bits, and the compilation of metadata required during the conversion upfront. Moreover, general optimisation principles such as buffered I/O, streaming processing, and reuse of memory buffers are applied throughout.

In Chapter 4, we find that the proposed approach achieves linear time complexity for purely row-wise mappings and bounded memory usage for all mappings, including those that require cross-file lookups. This is backed up empirically by running the tool on differently scaled synthetic feeds as well as a large real-world feed. A conversion speed of up to 1.5 million triples per second could be observed

for non-trivial mappings.

We conclude in Chapter 5 that goals set out have been achieved, and that the tool successfully fills a gap in the landscape of GTFS-to-RDF conversion, albeit with room for improvement. A key future improvement is the decoupling of the mapping from the build phase, which would allow users to specify the mapping in a more flexible and hence more user-friendly way.

Chapter 2

Background and Related Work

In this chapter, we will provide the necessary background on GTFS, RDF, and SPARQL to understand the problem at hand and the proposed solution. We will also further motivate the need for an efficient and flexible GTFS-to-RDF converter as well as briefly introduce related tools and conversion approaches. Finally, we will condense the key insights gained into a set of requirements and contributions this work sets out to achieve.

2.1 GTFS

A transit data schedule feed per the GTFS reference [1] is merely a ZIP archive containing a set of plain-text CSV files. Figure 2.1 shows a typical subset of these files and how they reference each other via identifiers.

GTFS defines both required and optional files. In Figure 2.1, the solid boxes illustrate core files that appear in most feeds, whereas the dashed boxes represent optional files that may or may not be present in a feed. Each of these files has a specific structure and a set of fields that must be present as well as a set of optional fields. For example, the core chain `routes.txt` → `trips.txt` → `stop_times.txt` in Figure 2.1 is connected via required identifiers, while additional columns such as `shape_id` may be present or absent. Some files may only be required if certain other files are present (or not present) or if certain fields from other files are present (or not present). The same holds for fields within files. Some fields may only be required if certain other fields are present (or not present). This is exemplified in Figure 2.1 by the conditional link from `trips.txt` to `shapes.txt`,

which only matters when trips actually reference a `shape_id`, and by the calendar information referenced via `service_id`.

Furthermore, there are a number of rules that govern the values of fields and the interplay of values across files. For example, if a field contains an ID that references another file, then the value of that field must match the value of a corresponding field in the referenced file. In Figure 2.1, this means that each `trip_id` occurring in `stop_times.txt` must correspond to a `trip_id` defined in `trips.txt`, and each `stop_id` must refer to an entry in `stops.txt`. Moreover, fields are specified to have a certain data type, and the values of these fields must conform to that type's formatting. For instance, coordinates such as `stop_lat` and `stop_lon` are numeric, and date fields such as `start_date` and `end_date` in `calendar.txt` must be strictly in the format `YYYYMMDD`.

While most files contain information that specifies the semantics of the transit network, some files contain metadata, such as `feed_info.txt`, that describe the feed itself, e.g. by providing a name, version, publisher information, main language used, etc.

Hence, a GTFS feed is quite simple in its basic structure. However, it can become fairly complex in its details and the interplay of different files, as can be seen in the GTFS reference graph in the Appendix, Section 3.

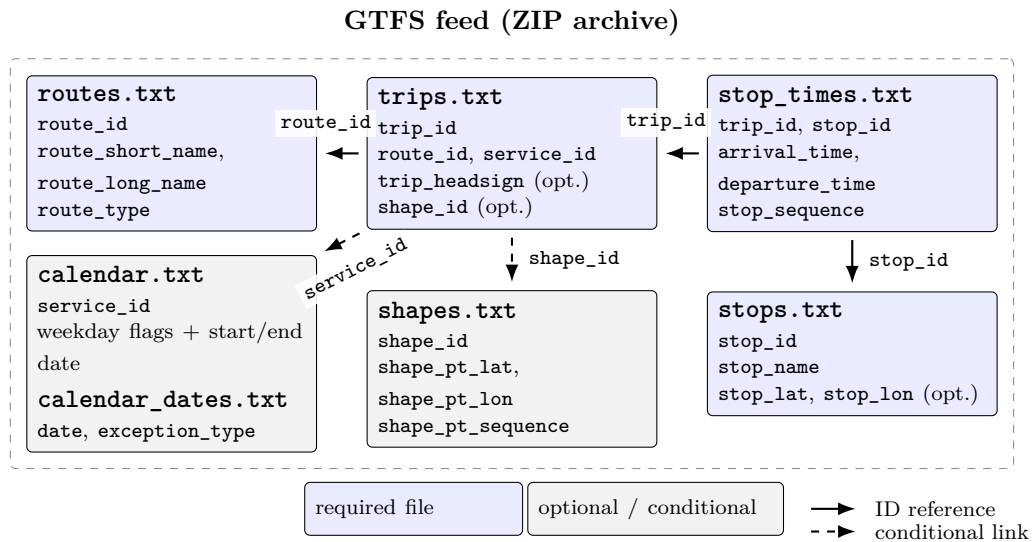


Figure 2.1: A GTFS feed is a ZIP archive containing interrelated CSV files. Arrows indicate dependency direction ($A \rightarrow B$ means that file A references file B). Solid arrows denote identifier-based dependencies, while dashed arrows indicate conditional dependencies on optional files, such as service information via `service_id` and path information via `shape_id`.

2.2 RDF

The graph-based RDF data model [4] is founded on the concept of triples, which consist of subject, predicate, and object, not unlike a simple statement or fact in human language. These triples can be visualised as directed edges in a graph, where the subject and object represent nodes and the predicate represents the directed edge connecting them. Datasets adhering to RDF form so-called knowledge graphs.

The components of a triple can take various forms:

- **IRI**: an IRI (*Internationalized Resource Identifier*) is a unique identifier that can be used to unambiguously identify resources in the RDF graph (e.g. `http://example.org/bus_stop_A`).
- **Blank Node**: a blank node is an anonymous resource that does not have an IRI but can be used to attach properties to a resource without explicitly naming it.
- **Literal**: a literal is a value such as a string, number, or date (e.g. "Central

Station", 42, "2024-06-01"). Literals can further be typed with a data type.

- **xsd:integer**: an integer value (e.g. "42"^^xsd:integer).
- **xsd:date**: a date value in the format YYYY-MM-DD (e.g. "2024-06-01"^^xsd:date).
- **xsd:string**: a string literal (e.g. "Central Station"^^xsd:string) with an optional language tag (e.g. "Central Station"@en).
- **geo:wktLiteral**: a spatial geometry represented in Well-Known Text (WKT) format (e.g. "POINT(7.8352 48.0128)"^^geo:wktLiteral) [8, 9].
- Many more data types exist, and custom data types can also be defined as needed.

Note that each one comes with a specific syntax and formatting rules that must be followed for the RDF data to be valid. It becomes apparent that such knowledge graphs can be integrated with other graphs very easily by simply using the same IRIs to refer to the same entities, or by using properties to link entities across graphs.

RDF data can be expressed in various serialisation formats, such as Turtle, N-Triples, or others, which provide different ways to represent the triples in a text-based manner. Turtle is a popular and human-readable format, allowing for compact representation by means of syntax shortcuts and prefixes (i.e. namespace abbreviations), whereas N-Triples is an explicit format and, hence, more verbose.

Example. Consider the following small set of transit information facts:

1. The stop *Bertoldsbrunnen* has the name “Bertoldsbrunnen”.
2. The trip *trip_4711* belongs to the route *Linie 1* and has the destination sign “Landwasser”.
3. The trip *trip_4711* serves the stop *Bertoldsbrunnen*.

A possible text-based RDF representation (Turtle) is shown below:

```
@prefix ex: <http://example.org/> .
@prefix gtfs: <http://vocab.gtfs.org/terms#> .

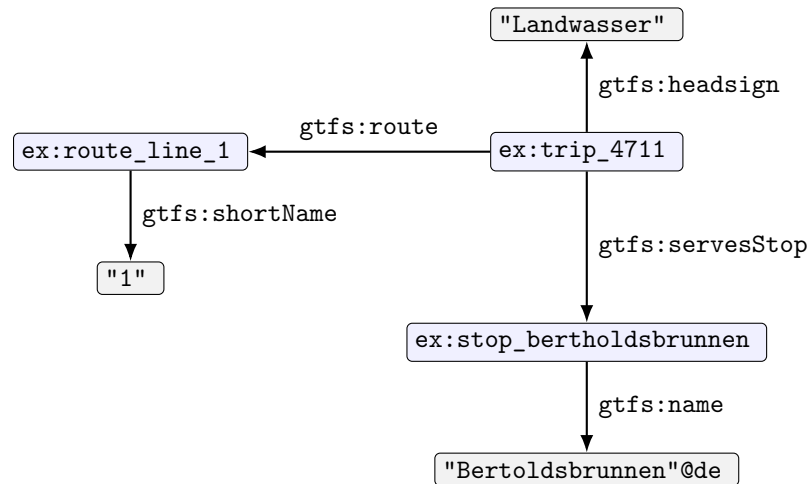
ex:stop_bertholdsbrunnen gtfs:name "Bertoldsbrunnen"@de .
ex:trip_4711 gtfs:route ex:route_line_1 .
```

```

ex:trip_4711 gtfs:headsign "Landwasser" .
ex:trip_4711 gtfs:servesStop ex:stop_bertholdsbrunnen .
ex:route_line_1 gtfs:shortName "1" .

```

Or equivalently as a graph:



If we wanted to express one of those triples using N-Triples, it would look like this:

```

# Turtle (requires prefix declaration alongside -> emitted for brevity)
ex:trip_4711 gtfs:route ex:route_line_1 .

# N-Triples (prefixes resolved)
<http://example.org/trip_4711> <http://vocab.gtfs.org/terms#route> <->
  <http://example.org/route_line_1> .

```

2.3 SPARQL and SPARQL Endpoints

SPARQL (*SPARQL Protocol and RDF Query Language*) [5] can be used to query RDF data. SPARQL engines such as *QLever*, developed at the University of Freiburg, can index RDF data, generate query plans from SPARQL queries, and thereby enable efficient search and data retrieval on these knowledge graphs [6].

An example SPARQL query that retrieves the name of each stop served by the trip *trip_4711* might look like this:

```

PREFIX ex: <http://example.org/>
PREFIX gtfs: <http://vocab.gtfs.org/terms#>

SELECT ?stopName WHERE {
  ex:trip_4711 gtfs:servesStop ?stop .
  ?stop gtfs:name ?stopName .
}

```

For our specific use case of transit data, there are powerful tools in the *QLever* ecosystem that facilitate insights and meaningful applications on GTFS data. For example, *pfaedle* is a tool that can generate precise trip geometries for GTFS feeds by using OSM (*OpenStreetMap*) data [2, 10]. Once converted to RDF, this enriched data can then be processed and queried by *QLever*. By employing the *QLever* middleware *qllever-petrimaps* in a further step, one can then efficiently visualise the results of SPARQL queries on maps and thereby gain valuable insights into the transit network and its interplay with the urban environment [11]. Figure 2.2 shows an example of such a visualisation, where query results are interactively rendered as map overlays.

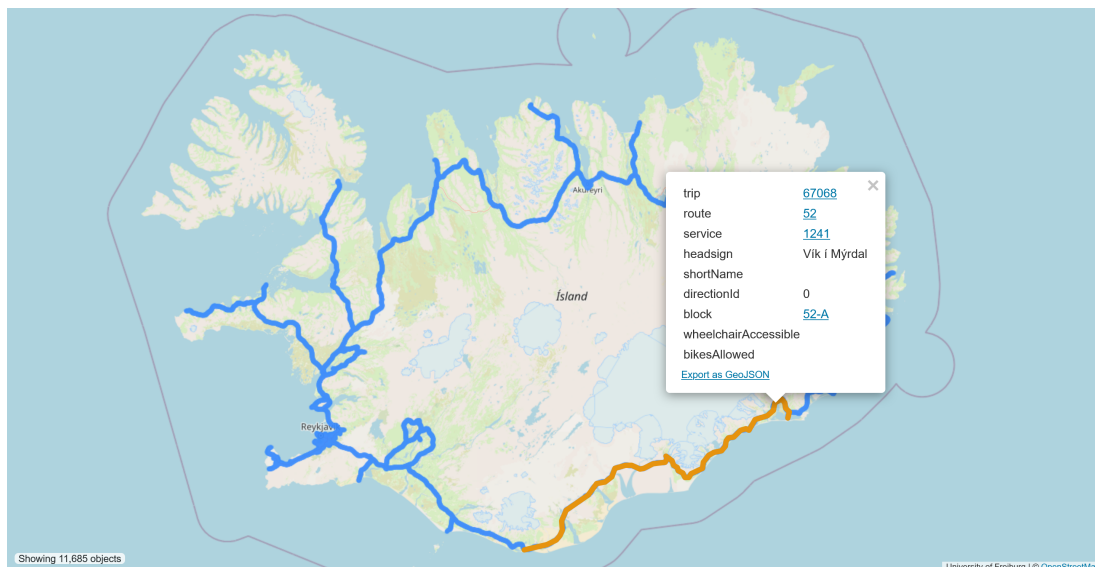


Figure 2.2: Example visualisation of SPARQL query results in *qllever-petrimaps*: a map of Iceland showing all trips operated by the bus company *Strætó bs* with further information [12].

2.4 Related Converters

During research, it became clear that there are not many tools that directly address the problem of converting GTFS data to RDF. In this section, we will briefly discuss the projects found as well as some related tools that, while not specifically designed for GTFS, can be used for similar purposes. From this, we can get an overview about the functionalities that related state-of-the-art tools offer, and identify gaps that our tool can fill.

2.4.1 GTFS-to-RDF Converters

There is a tool called *gtfs-csv2rdf*, developed in Node.js as part of the *OpenTransport* project, that converts GTFS data to RDF [13, 14]. It processes the GTFS files sequentially and produces a fixed set of triples for each row per file, based on a predefined mapping. This streaming approach allows it to handle large feeds without consuming too much memory. It also takes rudimentary care of type-format mismatches between GTFS and RDF, e.g. by converting date fields to the appropriate `xsd:date` format. However, there are typos in the source code (e.g. ‘Tuesday’ instead of ‘TuesDAY’) which will produce invalid or incomplete conversions, and the project is fairly old (latest commit in 2018) and seems to be no longer maintained. Also, this very linear pipeline approach does not allow for any joining of information across files, which might be desirable for converting relational data like GTFS to a graph-based model like RDF.

All in all, this simplistic tool provides a good baseline for GTFS-to-RDF conversion, while lacking important features and maintenance.

2.4.2 Generic CSV-to-RDF Converters

There is a number of generic CSV-to-RDF converters. In a comparative blog article, *Tarql* was described as a very good tool for this purpose ‘because of its performance and usage of SPARQL as a mapping language’ [15].

Tarql is a command-line tool that is designed to be flexible and can handle a wide variety of CSV formats, making it suitable for converting GTFS data to RDF file-wise. Mappings from CSV files to RDF can be constructed using expressive

SPARQL CONSTRUCT queries. This is a powerful feature that comes with a collection of built-in functions for data transformation and manipulation, which can be very useful for handling the intricacies of GTFS data [16].

In summary, *Tarql* is a very capable tool for the job of converting CSV to RDF. GTFS being a ZIP archive of interrelated CSV files, however, would require a more complex pre-processing pipeline to be fully processed by *Tarql*.

2.4.3 RML/KGC Engines

The *MoIn* project set out to link GTFS data with Wikidata Geospatial data [17, 18]. In the process, they identified the need to transform data during conversion, e.g. by normalising time fields to a consistent format: In GTFS, times may exceed 24 hours to indicate operations past midnight, whereas in RDF, time literals must be in the format `hh:mm:ss` and cannot exceed 24 hours [1, 18]. For conversion to RDF, the RMLMapper engine was used, which executes mappings defined in RML [19].

RML (*RDF Mapping Language*) is a powerful and flexible language for defining mappings from various data formats (including CSV) to RDF [20]. RML is very portable and widely used: There are quite a few engines that can execute RML mappings and thereby perform conversions, such as the Java-based *RMLMapper*, the streaming-oriented *RMLStreamer*, and the Python-based *Morph-KGC* [19, 21, 22].

In a proposal for a preprocessor to RML engines to tackle functionality and performance deficits of these, the above mentioned engines were evaluated on the GTFS-Madrid benchmark [23, 24]. Specifically, a mapping was tested where a trip node (`trip_id` in `trips.txt`) references a service node (`service_id`) only if this service exists in its parent source (i.a. in `calendar.txt`), as is illustrated in the following Turtle snippet:

```
@prefix ex: <http://example.org/> .
@prefix gtfs: <http://vocab.gtfs.org/terms#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

# only emit this triple if service_id=1241 exists in calendar(.txt) or
# calendar_dates(.txt):
```

```
ex:trip_4711 gtfs:service ex:service_1241 .

# the service resource itself originates from the calendar files:
ex:service_1241 gtfs:startDate "2024-10-01"^^xsd:date .
ex:service_1241 gtfs:endDate "2024-12-14"^^xsd:date .
```

The authors found that RMLMapper and RMLStreamer did not produce any results after an hour of execution. Hence, one would need to incorporate these into a more complex pipeline with pre-processing, if such validity checks are desired. *Morph-KGC*, however, was able to produce results for both unscaled (i.e. same size as published) and scaled version of the Madrid dataset, finishing the unscaled version in merely 11s. This makes *Morph-KGC* a strong contender for GTFS-to-RDF conversion [23, 24].

2.5 Requirements and Contributions

From existing converters, we can see that certain features are desirable for GTFS-to-RDF conversion, if their implementation is feasible. For example, a **streaming approach** that can process the GTFS files sequentially and produce RDF triples on the fly is desirable to handle large feeds without consuming too much memory [13]. For ease of use, it is beneficial if the tool can run **directly on GTFS ZIP archives** without any pre-processing [13]. Moreover, it should enable **flexible mappings** from GTFS to RDF without requiring changes to the engine or source files, as supported by declarative mapping approaches such as *Tarql* and RML-based engines [16, 20]. In addition, GTFS being a multi-file dataset, a converter should support **cross-file persistence**, i.e. the ability to access data across files and resolve references (e.g. considering a certain row from `stop_times.txt` only if it references an existing `stop_id` from `stops.txt`) [1]. Furthermore, the need to **transform data** during conversion, e.g. to accommodate type-format mismatches between GTFS and RDF, became apparent in practice [13, 18].

Practical downstream integration with tools such as *QLever* and *qllever-petrimaps* further motivates additional requirements, as became clear during discussions with my supervisor. In particular, **aggregation** of information from several GTFS rows into a single triple (e.g. producing a single `LINestring` [25] trip geometry from shape points) facilitates map-based visualisation via *qllever-petrimaps*. Also,

if one wants to query for trips that operate on a certain date, it is useful to have service dates already materialised explicitly, instead of having to derive them at query time. This motivates a controlled form of **fan-out**, i.e. emitting multiple triples from a single input record: `calendar.txt` specifies `start_date` and `end_date` for a service, but not the individual operation dates required for direct date-based querying [1].

In the following chapter, we present the design and implementation of a tool that incorporates the above features for fast and flexible GTFS-to-RDF conversion, filling gaps of existing software.

Chapter 3

System and Implementation

In this chapter, we will explain the design and implementation choices of `gtfs2rdf` made to achieve the goals we identified in previous chapters, above all customisability and efficiency. We will approach this by looking at the user interface and configuration options first, and then take a closer look at the parts of the software that enable these features and how they are designed for efficient execution.

3.1 Overview and Design Goals

The converter is designed to be a command-line tool that takes a GTFS feed ZIP file as input and produces RDF output in either Turtle or N-Triples. Further user parameters involve diagnostic options to get insights into the conversion process and for debugging as well as modifying RAM usage to cater to limited resources. Lastly, a pre-run option is available where user-extensible mappings are validated and statistical estimates concerning output and conversion process are provided without actually writing output. This is recommended for detecting bugs or errors that might otherwise appear several minutes into the conversion process.

Since it is not a converter's task to ensure input correctness, the implementation follows a GIGO (*garbage in, garbage out*) principle. Hence, faulty input can lead to faulty output. However, provided that the input data and user-defined mappings are valid, output will be syntactically correct RDF. At the same time, users themselves can implement quite a few checks, spotting GTFS feed issues and inconsistencies (see Subsection 3.4.3).

For a complete view of available command-line options, refer to the Appendix,

Section 1.

Processing overview. Figure 3.1 summarises the processing phases and the main internal components we will encounter in more detail in the following sections.

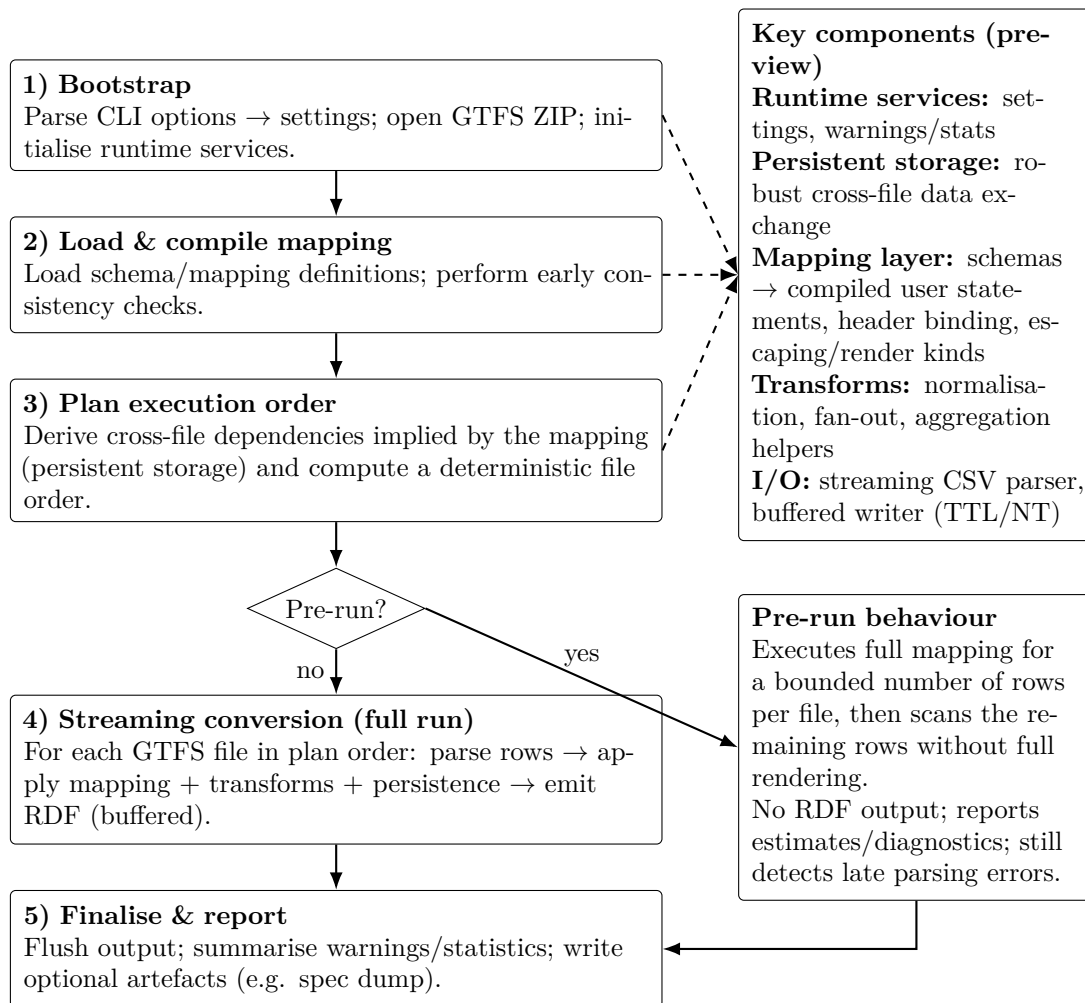


Figure 3.1: High-level processing overview of `gtfs2rdf`. The main pipeline on the left consists of the core processing phases, while the right side provides a preview of key internal components that will be detailed in the following sections.

3.1.1 Core Concepts and Terminology

The following terms are used throughout this chapter:

Term	Role	Implementation form
schema	per-file mapping unit	object of class <code>Schema</code>
user statement	user-defined mapping rule	entry in <code>TRIPLES</code> or <code>NO_WRITE_INSTRUCTIONS</code>
instruction	executable compiled statement	object of class <code>Instruction</code>
transform	value-processing function in the DSL	<code>Transform2One</code> or <code>Transform2Many</code>
storage kind	persistent storage abstraction	<code>VARIABLE</code> , <code>MULTI_MAP</code> , <code>TUPLE_MAP</code>

3.2 Mapping and DSL

We will now describe the core concept of per-file mapping units, so-called **schemas**. Each schema is represented by an object of C++ class `Schema`. A schema corresponds to exactly one GTFS file and defines how rows from that file are processed and converted into RDF.

To ensure mapping flexibility, a user can create their own schemas as a C++ module file and drop them into a designated folder. As a caveat, adding new ones currently requires rebuilding. A schema template will be provided along with the source code and a collection of example schemas for commonly used GTFS files. This modular and customisable approach may be particularly useful in case new GTFS files are added in the future, or if users want to create custom schema sets for different use cases (e.g. a lightweight set of schemas for quick conversion, or a more complex set with numerous cross-file dependencies etc.).

Furthermore, we will introduce a small DSL (*domain-specific language*), to be used within schemas, for defining how GTFS data should be transformed into RDF. This DSL allows users to specify mappings from GTFS CSV columns to RDF triples, including support for transformations, lookups, and persistence / storage access as we will see in the next sections. We will extend the DSL with the respective features as we go along. A complete reference of the final DSL syntax and semantics will be provided in the Appendix, Section 2.

A schema contains user-extensible declarative mapping rules, **user statements**. Each will be executed for each row of the corresponding GTFS file. Let us inspect a very simple and illustrative example snippet of a schema for the `stops.txt` file:

Listing 3.1: Example mapping for `stops.txt`

```
const std::unordered_map<std::string, std::string> PREFIXES = {
    {"stops", "https://gtfs.org/stops/"},
    {"geo", "http://www.opengis.net/ont/geosparql#"},
    {"rdf", "http://www.w3.org/1999/02/22-rdf-syntax-ns#"},
    {"gtfs", "https://w3id.org/gtfs2rdf#"};

const IRI SUBJ = IRI("stops", "{stop_id}");
const std::vector<Triple> TRIPLES = {
    {SUBJ, {"rdf", "type"}, {IRI("gtfs", "Stop")}},
    {SUBJ, {"gtfs", "stopName"}, {"{stop_name}", "de"}},
    {SUBJ, {"geo", "asWKT"}, {"POINT({stop_lon} {stop_lat})", ←
    IRI("geo", "wktLiteral")}}};
```

Note how a user would basically write triples that they want generated, utilising DSL expressions in curly braces, which we will call **placeholders**. In these placeholders, GTFS columns (e.g. `{stop_id}`) can be referenced. The respective value from the current row will be substituted at conversion time. Furthermore, note the (required) definition of prefixes at the top, enabling shorthand notation for IRIs. Note lastly that the object of a triple can be an IRI, a literal with an optional language tag or a literal with an optional datatype IRI.

If the resulting substitute value for a placeholder ends up empty (e.g. if a GTFS column is empty in the file), the whole respective triple will be skipped.

Note that all GTFS fields are treated as strings, the converter itself does not assume any data types anywhere. This gives great flexibility across various parts of the converter and users can implement their own type checks and conversions using transforms (see next Section 3.3) if needed.

At this point, it is worth stressing that GTFS-to-RDF conversion is rarely a pure one-to-one substitution of CSV fields. As we will see in the next sections, the embedded DSL is designed to express more complex operations declaratively, while the converter executes them in a streaming-friendly fashion.

3.3 Transforms

As was identified by the creators of *gtfs-csv2rdf* or the *MoIn* project, GTFS data often requires transformations in order to be useful in RDF form [13, 18]. To this end, we will introduce **transforms**, i.e. user-definable functions with a certain signature that draw from the richness of C++ and the performance optimisations of compiled code, while being able to be conveniently used in our mapping DSL.

GTFS column values can be piped through transforms in placeholders within the DSL, e.g. to normalise date formats, perform checks on values, or to fan-out a row into multiple triples. Two types are supported:

- **Transform2One**: produces one string output value from one or more input values (e.g. date normalisation)
- **Transform2Many**: produces multiple string output values from one or more input values (this will result in multiple triples being emitted that differ only in the respective placeholder with the **Transform2Many**)

Transforms can be chained together, such that values may undergo multiple transformations in a single placeholder. Furthermore, a small transform library will be shipped with the converter, containing transforms that may be useful or commonly needed. Two examples are **isInRange**, a **Transform2One**, that throws an error if a value is out of a specified range, and **isValidDecimal**, a **Transform2One**, that throws an error if a value cannot be parsed as a `xsd:decimal`. With this, we could extend our previous example (Listing 3.1) as shown:

```
// ... PREFIXES and SUBJ omitted for brevity ...
const std::vector<Triple> TRIPLES = {
    {SUBJ, {"rdf", "type"}, {IRI("gtfs", "Stop")}},
    {SUBJ, {"gtfs", "stopName"}, {"{stop_name}", "de"}},
    {SUBJ, {"geo", "asWKT"}, {"POINT({stop_lon} {stop_lat})", ←
    IRI("geo", "wktLiteral")}},
    {SUBJ, {"wgs", "lat"}, {"{stop_lat, \"-90\", \"90\" | isInRange ←
    | isValidDecimal}", IRI("xsd", "decimal")}}};
```

Note that **isInRange** takes three parameters by design (value to check, lower bound, upper bound) and `stop_lat` is taken from the `stop_lat` column of the current row and the bounds are literals (hence in quotes). If the value of `stop_lat`

passes the check, it will be passed to `isValidDecimal` to ensure it matches the `xsd:decimal` format that we specified as the datatype of the literal. Otherwise, a detailed error will be thrown (see Section 3.7).

Let us also examine an example snippet for `calendar.txt`:

Listing 3.2: Example use case for `Transform2Many`

```
// ... PREFIXES and SUBJ omitted for brevity ...
const std::vector<Triple> TRIPLES = {
  {SUBJ, {"gtfs", "serviceDate"}, {"{sunday, monday, tuesday, ↔
  wednesday, thursday, friday, saturday, start_date, end_date | ↔
  generateDates }", IRI("xsd", "date")}}};
```

This is a use case for a `Transform2Many`, where a date span (from `start_date` to `end_date`) is explicitly materialised into triples for each date that the service is operated on, further specified by the boolean weekday columns. Note that `generateDates` already takes care of proper date formatting internally.

With this extension, our DSL for placeholders in generalised form would look like this:

```
{ arg1, ..., argN | transform1 | transform2 | ... | transformM }
```

The arguments can be GTFS column values or literals, and the transforms are applied in a left-to-right order, i.e. `transform1` takes potentially several arguments, then `transform2` takes its output and so forth. One limitation with regards to `Transform2Many` is that they can only be used once per user statement. They can either take several arguments as input or their predecessor's output. Subsequent `Transform2Ones` will then be applied element-wise to each of the outputs of the `Transform2Many`.

For ease of use, users are able to define transforms in a schema using macros to then use them in the mapping DSL. For information on how to define transforms, please refer to the DSL reference in the Appendix, Section 2.

3.4 Cross-File Persistence

To power cross-file lookups and references in the mapping, we will introduce a persistent storage component that allows schemas to store and retrieve values across the processing of different GTFS files.

Three storage kinds are supported:

- **VARIABLE**: a key \mapsto value store
- **MULTI_MAP**: a key \mapsto values store
- **TUPLE_MAP**: a key \mapsto tuples store, where a tuple is a collection of values

Variables are meant for recurring lookups of the same value across files or to cache values between two user statements of the same schema. Let us look at an example of the former case:

```
// feed_info.txt
const std::vector<std::string> NO_WRITE_INSTRUCTIONS = {"{feed_lang > ←
    FEED_LANG@feed_info.txt}"} // stores the feed language in variable ←
    FEED_LANG

// stops.txt
const std::vector<Triple> TRIPLES = {
    {SUBJ, {"gtfs", "stopName"}, {"{stop_name}", ←
    "{FEED_LANG@feed_info.txt}"}}};
```

With the mapping above, we can dynamically fill in the language tag everywhere we want it, drawing from the value of `feed_lang` in `feed_info.txt`. A few DSL-related things to note here: schemas support storage-only user statements that are only strings and which do not lead to triple emission. `>` is used to denote a write operation to storage and `@` is to be used in a placeholder to denote the context of a storage access. This is required to make the dependency visible such that files can be read in correct order, which we will cover in more detail in Subsection 3.4.3.

Multi-maps and tuple-maps, on the other hand, are intended as mass storages. Let us look at an example of multi-maps:

```
// calendar_dates.txt
const std::vector<std::string> NO_WRITE_INSTRUCTIONS = {
```

```

"{ service_id : date, exception_type | isDisabledDate | ←
  convertDate2xs_unchecked > disabled_dates@calendar_dates.txt }"};

// calendar.txt
const std::vector<Triple> TRIPLES = {
  {SUBJ, {"gtfs", "serviceDate"}, {"{sunday, monday, tuesday, ←
  wednesday, thursday, friday, saturday, start_date, end_date, ←
  service_id | generateDates@calendar_dates.txt }", IRI("xsd", "date")}}};

```

Note that the use of `:` denotes a keyed storage write. Hence, `:` can never appear without `>` in a placeholder. In the example above, we store each date where a given service is disabled, which is the output of `isDisabledDate`, properly formatted in a multi-map, keyed by `service_id`. Therefore, this multi-map is of the form `service_id ↦ [disabled_date1, disabled_date2, ...]`. Then, in `calendar.txt` we can read from multi-map `disabled_dates` in the context `calendar_dates.txt` to exclude explicitly disabled dates for a service from the date range generated by `generateDates` (extending the example Listing 3.2 from before).

This also implies that transforms need to have a storage handle. For information on how to access storage from transforms and for more details on storage semantics, please refer to the DSL reference in the Appendix, Section 2.

To summarise, let us update our generalised DSL form to include storage operations:

```

{ arg1, ..., argN [: argN+1, ... argN+0] | transform1 | transform2 | ←
  ... | transformM > STORE_NAME@CTX }

```

Here, the part in brackets is optional and separates keys from values for storage writes. Both will be presented to `transform1` as arguments in that order.

3.4.1 Aggregation

Using this storage setup, we can also realise the aggregation functionality that we identified as a major requirement for GTFS conversion. Let us look at an example of this:

```
// shapes.txt
const std::vector<std::string> NO_WRITE_INSTRUCTIONS = {"{ shape_id : ↵
    shape_pt_sequence, shape_pt_lon, shape_pt_lat > shapes@shapes.txt }"};

// trips.txt
const std::vector<Triple> TRIPLES = {{IRI("gtfs2rdfgeom", ↵
    "shapes_{shape_id}"), {"geo", "asWKT"}, {"{"{shape_id | ↵
    getLinestring@shapes.txt"}", IRI("geo", "wktLiteral")}}};
```

In this example, we store all shape points along with their order index `shape_pt_sequence` for each `shape_id` from `shapes.txt` in storage. The resulting tuple-map will have format: `shape_id ↦ [(shape_pt_sequence, shape_pt_lon, shape_pt_lat), ...]`. Then, in the `trips.txt` schema, we can read all tuples for a given `shape_id` inside `getLinestring`, a `Transform2One`. This transform can use some sorting algorithm (e.g. from standard library) to order the shape points by `shape_pt_sequence` and concatenate them into a WKT linestring, which is then emitted as the object of a triple. To reduce output size, we can also ensure that each shape geometry is emitted only once by creating a separate geometry node (note `gtfs2rdfgeom`) and letting trips reference that node, instead of attaching the geometry directly to each trip. A slim multi-map can be used to record which shape IDs have already been materialised, so that geometry nodes are only generated for shapes not yet seen.

3.4.2 SQL-backed Storage

Since `shapes.txt` may easily be several gigabytes in size, it might be infeasible to store the whole tuple-map in memory for large feeds. There may also be other schemas adding to memory demands with their own storage needs. To this end, multi-maps and tuple-maps are handled by an embedded SQL database (SQLite) to allow for disk-backed storage, while variables are still kept in memory for fast read and write access.

The *SQL Amalgamation* is used to integrate SQLite into the converter as a single `.h` and `.cpp` file, which are then compiled together with the rest of the codebase. Hence, no external dependencies are imposed on users when building the converter. At the same time, SQLite is a widely used, mature, and performant embedded database engine, which makes it a good fit for our use case [26].

Each multi-map and tuple-map is represented by a separate lazily created table in the database. The tables are indexed with `(key, val)` and `(key, val1, val2, ...)` respectively as primary keys to ignore duplicates cheaply and ensure fast lookups. Since our DSL allows multiple keys, these are concatenated internally to form a single key. For tuple-maps, it is ensured that the tuple arity (e.g. `(seq, lon, lat)` has arity 3) is consistent across writes. Hence, a fixed number of value columns is created for each table.

For performance reasons, prepared statements are used for all database operations, avoiding SQL parsing overhead and allowing for efficient execution. Furthermore, transactions are performed in batches instead of committing each insert separately to reduce disk I/O. Moreover, the database is configured to operate in a fixed heap size budget (soft / hard limits) to avoid out-of-memory errors and to allow users to control RAM usage, trading cache size (lookup speed) for memory efficiency as needed.

Overall, this enables the converter to scale to very large feeds, because RAM usage remains bounded and controllable even when large amounts of intermediate state need to be stored persistently on disk.

3.4.3 Topological Sorting

As established, cross-file dependencies can be realised by storage operations in the mapping. However, for a single-pass streaming execution, it is crucial to ensure that files are processed in an order that respects these dependencies. To resolve this, we will perform a topological sort on the schemas at runtime based on the dependencies implied by storage reads. Topological sorting is a common algorithmic technique for ordering tasks with dependencies, e.g. represented as a directed graph. Hence, correct usage of context hints (e.g. `@feed_info.txt`) is crucial for this graph to be constructed correctly. In our particular tool, an implementation of Kahn’s algorithm is used to compute a valid file processing order [27]. Cycles cannot be handled and will be reported as errors.

Additionally, two micro-optimisations are in place here: first, storage writes from schemas that have no dependents are disabled to save unnecessary database operations / disk usage. Second, once all dependents of a schema have been processed, the respective storage is cleared to free up disk space.

Optional integrity checks. While `gtfs2rdf` follows a GIGO principle, the absence of circular dependencies in the GTFS reference graph (Appendix, Section 3) means that a strictly ordered single-pass run can, in principle, implement comprehensive reference checks (e.g. rejecting dangling `stop_id` or `service_id` values) without requiring multiple passes. Such checks are therefore technically expressible in the mapping layer should the user so wish.

3.5 Schema Lifecycle

In Figure 3.1, this section corresponds to Phase 2 (Load & compile mapping) and Phase 3 (Plan execution order).

A schema’s lifecycle basically consists of four phases: initialisation, compilation, binding, and execution. We will cover the first three in this section, and the execution phase in the next one, since it involves a lot of streaming execution details that deserve their own section.

3.5.1 Initialisation

The tool first scans the input ZIP archive to determine which GTFS files are present. Only schemas corresponding to those files are then initialised, avoiding unnecessary work. Initialisation of a schema involves instantiating an object of the C++ class `Schema`. This object stores, among other things, the name of the corresponding GTFS file (e.g. `stops.txt`), the defined prefixes, and the user statements (`TRIPLES` and `NO_WRITE_INSTRUCTIONS`) for further use.

Also, these user statements will already be written out into raw strings during initialisation, to be parsed and processed in the next phases, with prefixes either already expanded (for N-Triples) or retained in prefixed form (for Turtle). This is part of the general strategy of this converter to materialise all static parts of the triples before conversion time to reduce runtime overhead. For each placeholder of these raw strings, escape information is stored alongside in order to later escape its evaluation result according to RDF term kind (IRI/literal), datatype or language tag, and the chosen output format.

3.5.2 Compilation

During the compilation phase, the raw strings produced in the initialisation phase are cut into segments of static and dynamic parts (i.e. placeholders), where the latter are parsed such that all information required for conversion is extracted and stored alongside, such as:

- **Argument resolution:** distinguishing CSV column references, variable lookups, and literal values
- **Transform chain:** attaching required transforms
- **Storage metadata:** deriving key/value partitioning and storage information (i.e. what is stored where?)
- **Context extraction:** deriving context information for storage reads to make dependencies visible

This way, during triple rendering, the converter can just write static parts directly to output and only needs to resolve placeholders by applying the stored information and performing the required operations. After the compilation phase, the acquired dependency information is resolved to compute a valid file processing order for upcoming stages (see Subsection 3.4.3).

3.5.3 Binding

Now that the execution plan is set, all files are processed sequentially in the computed order. For each file, the header row is read first to bind column names to indices for fast access during conversion. That means that user statements are finalised as executable objects of class `Instruction`, containing static string segments and dynamic segments with column index references, attached transforms, and storage metadata. Note that instructions are only marked valid for user statements that either (i) reference no CSV columns, or (ii) only reference columns that are present in the observed header. All other instructions will be ignored downstream to reduce overhead.

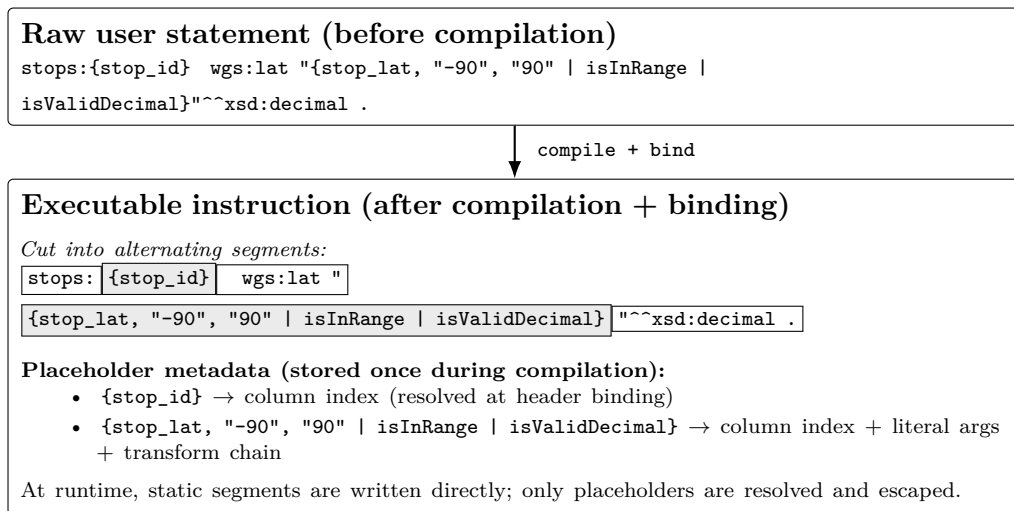


Figure 3.2: Compilation splits a user statement into alternating static segments and placeholders. Placeholders carry pre-parsed metadata, so runtime work is limited to resolving and escaping dynamic parts.

These instructions are then fit for streaming execution, which we will cover in the next section.

3.6 Streaming Execution: Parsing, Rendering, Writing

This section corresponds to Phase 4 (Streaming conversion) and the I/O component in Figure 3.1.

We will now take a closer look at the streaming execution phase, where the actual input reading, RDF rendering, and output writing happen. This phase is crucial for the converter’s performance and memory efficiency and involves several optimisations to achieve these goals. The process for each schema is the same, hence, we will describe it in general terms for a single file, which is then repeated for all files in the computed order.

3.6.1 Per-file execution flow

The data flow for a single file can be summarised as shown in Figure 3.3:

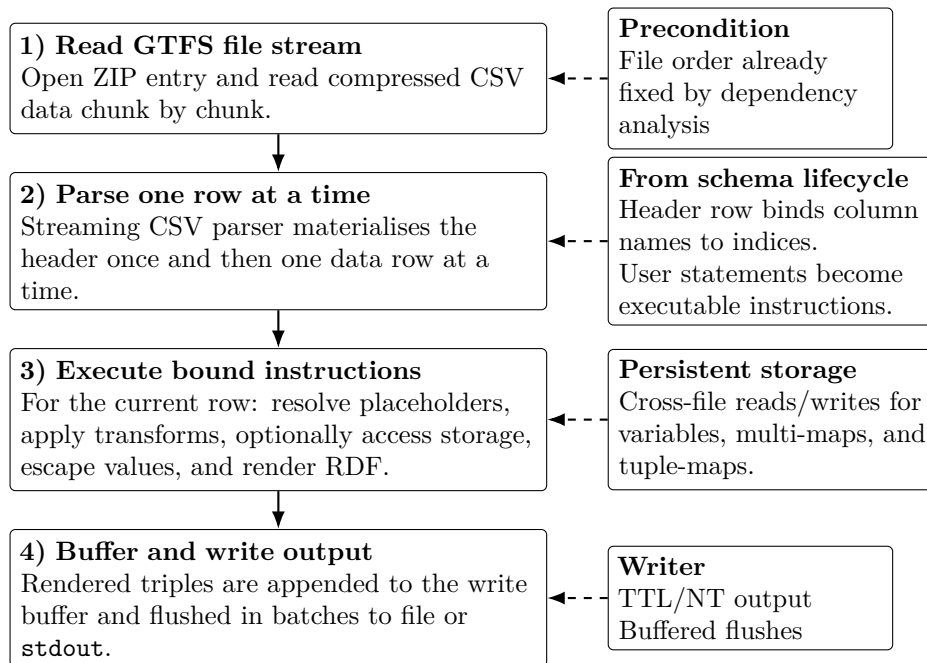


Figure 3.3: Per-file streaming execution in `gtfs2rdf`.

This procedure is repeated until all rows of the input file have been processed, whereupon the converter moves on to the next file in the execution plan.

Note that this is a single-pass row-driven execution model, where only one row is fully materialised in memory at a time, and triples are emitted immediately after processing each row. This allows for a very low memory footprint, as the converter does not need to hold large amounts of data in memory, as well as very fast processing. We will examine details and optimisations of the different stages in the following subsections.

3.6.2 Streaming CSV Parsing

The converter uses a custom streaming CSV parser that reads the input file in chunks and parses rows one at a time. Parsing is state-based to handle cases where rows may span across chunk boundaries. That means that the parser maintains an internal state machine to keep track of whether it is currently inside a quoted field, whether it has encountered a newline character, etc. With this approach, each character is appended to an internal buffer for the current CSV field until the next field is encountered, at which point the field is finalised and added to

the current row buffer. Once a newline is encountered outside of quoted fields, the row is finalised and passed on to the rendering stage.

Optimisations applied in the parser include:

- **Buffered reading:** input file read in chunks.
- **In-place parsing:** characters are processed directly from the input buffer and copied exactly twice (once to field buffer, once to row buffer) without intermediate structures.
- **Reuse of buffers:** field and row buffers are reused, all files share the same read buffer.

This design allows for effectively bounded memory usage (always one row + constant number of buffers in memory), few peak allocations, and fast parsing.

3.6.3 Triple Rendering

The row buffer populated by the parser is passed to each executable instruction of the schema. Each instruction controls a number of buffers on its own:

- **Output buffer:** for the resulting triple(s), reference passed to writing stage after rendering
- **Argument buffer:** for storing the arguments needed for transforms (e.g. column values, literals)
- **Temporary swap buffers:** for intermediate results of transform applications
- **Transform2Many result buffer:** Transform2Many produces multiple outputs which are stored in a dedicated buffer

The rendering process for an instruction involves materialising the static segments of the triple and the dynamic parts (i.e. placeholders) alternately. Hence, this part directly exploits the representation introduced in Figure 3.2. For the dynamic parts, the argument buffer (which only stores references, not copies) is populated according to the placeholder’s metadata (e.g. column from row buffer, literal, storage variable) and the transform chain is applied to these arguments, where intermediate results are stored in the temporary swap buffers. Finally, the resulting value is escaped according to the escape information attached during schema initialisation and then appended to the output buffer. If a placeholder specified storage writes, the resulting value would be passed to the persistent stor-

age component according to the storage metadata (e.g. key/value partitioning, context). After all segments have been processed that way, the output buffer contains the final triple to be emitted.

In case of a `Transform2Many`, the process is similar. The key difference is that the transform’s output is stored in the dedicated `Transform2Many` result buffer. Every subsequent `Transform2One` in the transform chain is then applied element-wise to each of the outputs in that buffer. Finally, the output buffer is populated with a triple for each of the outputs of the `Transform2Many` where each triple only differs in the respective placeholder. Hence, the resulting output buffer contains the final triples to be emitted. A visualisation of this process is shown in Figure 3.4.

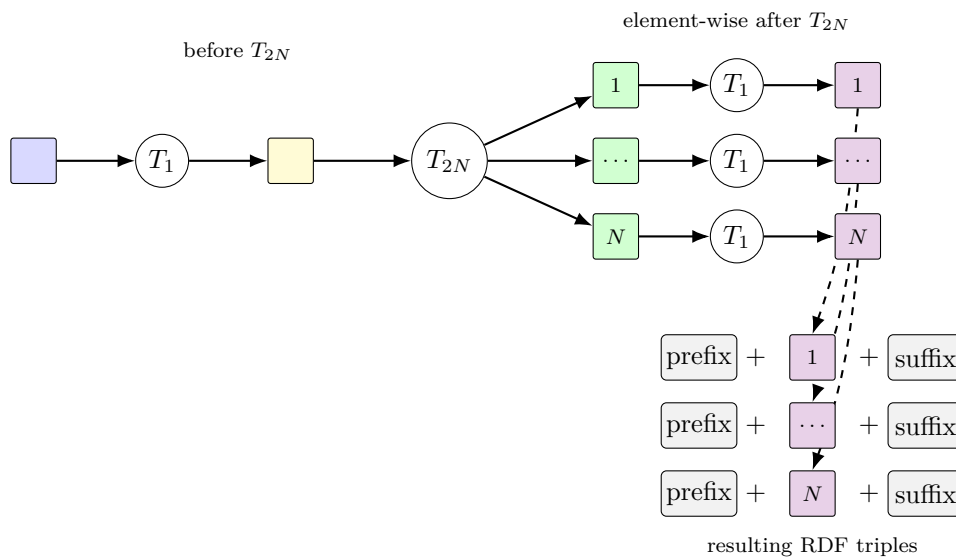


Figure 3.4: Schematic execution of a placeholder with transforms. A single value first passes through some `Transform2One` (T_1) transform. A subsequent `Transform2Many` (T_{2N}) transform expands it into values $1, \dots, N$. Any following T_1 transform is then applied element-wise to each of these values. Finally, each resulting value is combined with the same static prefix and suffix of the instruction, yielding several RDF triples that differ only in the respective placeholder component.

Optimisations applied in the rendering stage include:

- **In-place rendering:** static segments are written directly to output buffer, only dynamic parts require intermediate buffers.
- **Buffer reuse:** instruction-specific buffers are reused across rows.
- **Reference passing:** heavy use of references to avoid unnecessary copying

of data.

- **Early exit:** if a column value is empty or a transform result is empty, execution of current instruction will be skipped early.

Note that the early exit strategy also corresponds to the functionality to use transforms as filters, e.g. to exclude certain column values. This is a designated use case for transforms and embedded in the DSL. Further information can be found in the DSL reference in the Appendix, Section 2.

This design allows for very efficient rendering as the expensive placeholder parsing work is done entirely before the render loop. Therefore, rendering is a fairly linear process of writing static segments and resolving, transforming and optionally storing dynamic segments, all of which could not be done at a different stage. This phase is further streamlined by applying the abovementioned optimisations.

3.6.4 Output Writing

After rendering, the output buffer of an instruction contains the final triple(s) to be emitted. These are then directly copied to the output buffer governed by the writer module. Once the output buffer reaches a certain size threshold, it is flushed to the output target in a batch. This reduces system calls and disk I/O overhead. Both `stdout` and file output are supported.

3.7 Diagnostics

Since this tool offers a user-facing GTFS-RDF mapping interface, things can go wrong in various ways, such as DSL syntax errors or column name misspellings. Hence, a stack-based warning and error system is in place: As warnings or errors are emitted, they will bubble up in the program flow where context information is added. As a result, warnings / error messages contain precise information regarding the nature and the location of an incident. In particular, the schema parsing module (i.e. during Phase 2 in Figure 3.1) applies thorough syntax checks so that such errors do not only show up during the conversion phase several minutes into the program run.

Furthermore, the tool provides optional statistical output. The precise information

emitted depends on both a selected command line flag and whether the tool is executed in pre-run or full-conversion mode. In pre-run mode, reported figures are estimates based on a sample run. Among others, provided metrics include GTFS row and RDF triple counts. Detailed timing breakdowns are only available with dedicated build flags. Since this feature impacts runtime slightly but measurably, it is omitted from production builds so as to yield maximum throughput.

3.8 Testing

As a means to establish trust in conversion quality as well as to validate changes to the source code in the future, a small test suite is provided. It includes unit tests of dedicated functions such as RDF escape functions or the topological sorting algorithm as well as end-to-end tests: These will run a full conversion pipeline on a) a small curated feed and b) a larger real feed. The small feed was curated by hand to specifically exploit edge cases and tricky branches. Resulting output is exactly compared to the expected one, barring the order of triples. The real feed is intended for validation of the whole pipeline under more realistic conditions. For a quicker means of comparison, the number of output triples is compared to the expected number.

3.9 Efficiency Considerations

To summarise, phases 1 to 4 of our processing pipeline (Figure 3.1) utilise several engineering choices to optimise for performance and memory efficiency. To reiterate in concise form, these include:

- **Lazy schema initialisation:** only schemas corresponding to files present in the input ZIP are initialised.
- **Single-pass per file:** each file is read from start to end exactly once.
- **Chunked parsing:** input file is read in chunks, and rows are parsed one at a time.
- **Buffered writing:** output is buffered and written in batches to reduce I/O churn.
- **Reuse of temporary buffers:** buffers for parsing, rendering, and storage access are reused across rows.

- **Reference passing:** data is passed by reference where possible to avoid unnecessary copying.
- **Early exit:** if a triple would not be meaningful, execution of the respective instruction can be skipped early.
- **Precompilation before hot loop:** metadata already known at conversion time is precomputed, so that only necessary operations are performed and static parts can be written directly.
- **SQL-backed storage:** cross-file persistence is handled by an embedded SQL database to allow for disk spills.
- **Batched database operations:** prepared statements and batched transactions are used to reduce SQL overhead.
- **Storage access optimisation:** schemas without dependents do not write to storage, and once all dependents have been processed, the storage context is cleared.

Each choice alone will not make or break `gtfs2rdf`'s performance, but together they enable efficient conversion and support scalability on large feeds. We will evaluate this in more detail in the next chapter.

Chapter 4

Analysis and Evaluation

In this chapter, we will analyse the proposed approach from a theoretical and empirical perspective. The theoretical analysis will focus on the asymptotic time and space complexity of the streaming conversion pipeline, while the empirical evaluation will involve running the converter on synthetic and real-world GTFS feeds and measuring metrics deemed relevant for scalability and practical feasibility. Finally, we will briefly compare the approach to related tools to demonstrate that it fills a gap in the landscape of GTFS-to-RDF conversion.

4.1 Theoretical Analysis

We will only include the streaming conversion phase in the following asymptotic analysis, i.e. input reading, row-wise rendering, and output writing. Bootstrapping, schema preparation and execution planning depend only on the set of schemas. Seeing as at the time of writing, the GTFS reference only specifies 32 files, these phases are heavily dominated by the per-row conversion loop for any non-trivial GTFS feed [1]. Hence, they are negligible.

4.1.1 Time Complexity

Let B_{in} be the total uncompressed GTFS input size, B_{out} the total RDF output size, P the total number of executed placeholder evaluations, T_1 the total number of `Transform2One` applications, T_{2N} the total number of `Transform2Many` applications, and S the total number of persistent-storage accesses. Since trans-

form and storage costs are user-defined, we parameterise them by τ , τ_{2N} , and σ , respectively.

Under the assumption that ZIP DEFLATE decompression is linear in the amount of decompressed data, the streaming conversion phase runs in

$$O(B_{\text{in}} + P + \tau T_1 + \tau_{2N} T_{2N} + \sigma S + B_{\text{out}}). \quad (4.1)$$

Input reading and CSV parsing are linear in B_{in} , because the parser loads the input from start to finish in chunks and further work per-byte remains constant. Rendering is linear in the number of placeholder resolutions and transform applications, plus the number of output bytes. Output writing is linear in B_{out} , since emitted triples are copied to a write buffer once and flushed in batches.

For a fixed schema set, the quantities P , T_1 , T_{2N} , and S all grow linearly with the number of processed rows. Hence, overall runtime scales **linearly** with feed size, up to the user-defined costs of transforms and storage operations.

Non-linear costs. It is worth noting that transforms and storage operations can distort linear runtime locally. Referring back to Subsection 3.4.1, shape-to-`linestring` aggregation requires sorting of points grouped by `shape_id`. Depending on the number of points per shape, this can lead to super-linear runtime in the number of rows in `shapes.txt`.

4.1.2 Space Complexity

Peak memory usage during streaming is bounded by

$$O(b_{\text{read}} + b_{\text{field}} + b_{\text{row}} + I \cdot b_{\text{render}} + b_{\text{write}} + h_{\text{sql}}), \quad (4.2)$$

where b_{read} is the parser read-buffer size, b_{field} the current field buffer, b_{row} the current row buffer, I the number of instructions, b_{render} the instruction-local rendering buffers, b_{write} the write buffer size, and h_{sql} the heap bound of the temporary SQLite storage backend. Note that b_{read} , b_{write} , and h_{sql} are user-configurable (see the Appendix, Section 1) and will usually dominate memory usage. Hence, a user can control the memory usage fairly reliably.

Thus, memory usage is **bounded** independently of the total number of processed rows. Disk space usage of the persistent storage unit is not bounded, however, and may grow with the number of storage operations.

4.2 Experimental Analysis

4.2.1 Metrics

These are the main direct measurements that are collected during the following experiments:

- **Wall-clock time:** total end-to-end execution time
- **RSS:** maximum resident set size during execution, i.e. peak RAM usage
- **Input/output size:** uncompressed GTFS input and RDF output in bytes
- **Row count:** total number of rows parsed across all files
- **Triple count:** total number of RDF triples generated

4.2.2 Setup

All experiments are conducted on the same commodity machine with the following configuration:

Table 4.1: Experimental setup.

Component	Configuration
Execution environment	WSL2 on Windows
Storage	Internal SSD
Build configuration	Release build with in-depth timings enabled
Processor	AMD Ryzen 7 7840U with Radeon 780M Graphics, 8 cores, 16 logical processors, 3.30 GHz
Main memory	16 GB RAM (8 GB available to WSL2)
Memory measurement	RSS measured with <code>psrecord</code>
Sampling configuration	Interval: 0.5 s; child processes included
Monitoring command	<code>psrecord "[gtfs2rdf command]" --interval 0.5 --include-children --plot ram.png --log ram.log</code>
Read buffer	80 MB
Write buffer	80 MB
SQL Heap bound	500 MB
Output format	Turtle

4.2.3 Data

Table 4.2: Datasets used in the evaluation. Row counts exclude header rows.

Dataset	Scale	Uncompressed size [B]	Rows
Madrid-1	1	485 112	3 846
Madrid-10	10	4 913 037	38 460
Madrid-100	100	49 841 601	384 600
Madrid-1000	1 000	505 651 366	3 846 000
Madrid-10000	10 000	5 129 009 781	38 460 000
GTFS.de-real	–	5 893 951 972	133 562 578
GTFS.de-shuffled	–	5 893 951 972	133 562 578

The Madrid datasets were generated using the GTFS Madrid benchmark script [28], accompanying the benchmark described by Chaves-Fraga et al. [24]. However, the generated feeds are not fully GTFS-compliant and could therefore not be used unchanged for a full-featured evaluation. In particular, `calendar_dates.exception_type` contains the values 0/1 although the GTFS specification permits only 1/2; dates are encoded as `yyyy-mm-dd` instead of `yyyymmdd`; latitude and longitude

values lie outside the valid bounds; and `feed_lang` does not contain a valid language identifier. In addition, the generated `shapes.txt` file contains very densely represented geometries with approximately 1830 points per shape, whereas most shapes from the real-world German feed consist of around 200 points. Since shape aggregation requires sorting points per `shape_id` as well as heavy I/O operations for the temporary storage, we decided to omit shape aggregation for the Madrid runs. This is to investigate how the raw streaming pipeline scales with feed size.

For this reason, all Madrid runs were restricted to a reduced and further simplified schema subset consisting of `agency.txt`, `routes.txt`, `trips.txt`, `stops.txt`, `stop_times.txt`, `calendar.txt`, `calendar_dates.txt`, and `feed_info.txt`.

Mappings affected by invalid benchmark values were omitted where necessary. Consequently, the synthetic Madrid experiments evaluate only direct conversion and date materialisation, which covers relevant factors for the asymptotic behaviour of the streaming approach.

The full conversion pipeline, including shapes and additional validation and enrichment steps, is evaluated separately on the real-world `GTFS.de-real` dataset obtained from *gtfs.de*, as well as on a duplicate where rows in `shapes.txt` are shuffled randomly. Since *gtfs.de* provides its feeds in a processing-friendly and compact way, this will allow us to investigate the practical feasibility of our tool in both a favourable and a challenging scenario.

4.2.4 Results

Results in structured form are presented in Table 4.3. In the following, we will put the results into context and formulate findings with regard to the theoretical analysis from Section 4.1 and practical feasibility of the approach.

Table 4.3: Conversion results.

Dataset	Input size [B]	Rows	Triples	Time [s]	RSS [MB]
Madrid-1	485 112	3846	38 816	0.50	13
Madrid-10	4 913 037	38 460	389 990	0.51	16
Madrid-100	49 841 601	384 600	3 899 900	1.51	169
Madrid-1000	505 651 366	3 846 000	38 999 000	9.57	169
Madrid-10000	5 129 009 781	38 460 000	389 990 000	94.01	169
GTFS.de-real	5 893 951 972	133 562 578	295 187 794	196.94	623
GTFS.de-shuffled	5 893 951 972	133 562 578	295 187 794	1871.85	623

Scalability

When looking at the runtime behaviour for increasing scale of the Madrid datasets in Figure 4.1, we notice that runtime differs only slightly between Madrid-1 and Madrid-10. These runs are dominated by fixed costs of bootstrapping and schema preparation, which are independent of feed size. From Madrid-100 onwards, runtime grows approximately proportionally to input size, supporting the claim from the theoretical analysis.

To reason about space complexity, we look at the RSS measurements across the Madrid runs in Figure 4.2. Note that RAM demand grows up to a certain point after which it remains bounded. As stated in Table 4.1, read and write buffer size configurations amount to 160 MB in total, which approximates the observed bound of 169 MB well. Note that since the synthetic Madrid runs do not include shape aggregation, the temporary storage only becomes relevant for the real-world `GTFS.de-real` run. With `shapes.txt` being 3.3 GB in size, the temporary storage backend is heavily used and the peak size of the SQLite database on disk was 3.26 GB. Still, RAM usage never exceeded 623 MB, which is within the configured heap bound of 500 MB plus the read and write buffers. This supports the claim of bounded memory usage from the theoretical analysis.

Practicality

To speak to the practical feasibility of the approach, we look at the results for the real-world `GTFS.de-real` dataset. This feed is approximately 5.9 GB in size and covers schedule and trip geometry data for the entire German public transport

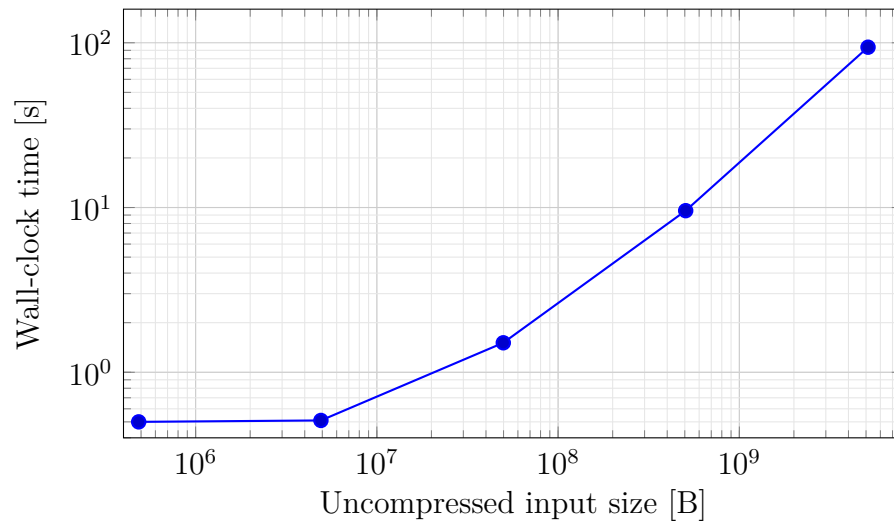


Figure 4.1: Wall-clock time over uncompressed input size for the synthetic Madrid datasets.

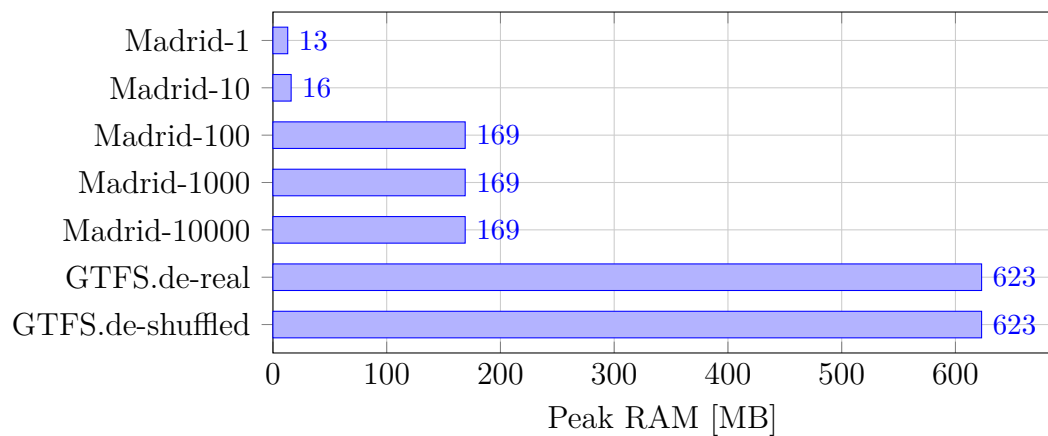


Figure 4.2: RSS across considered datasets.

network. The mapping included type checks for selected fields, such as dates, times, and latitude/longitude values, as well as date materialisation and shape aggregation: service dates from `calendar.txt` and `calendar_dates.txt` were combined and written explicitly, while points from `shapes.txt` were grouped by `shape_id` and rendered as linestrings.

The conversion of the unaltered feed completed in 3.3 min, required a peak RAM of 623 MB, and produced an RDF output of roughly 19 GB with 295 187 794 triples. This corresponds to a conversion speed of around 1.5 million triples / s for this particular mapping and feed. In contrast, the conversion of the shuffled feed took about 31.2 min, which is almost 10 times longer. This came to light only late in the thesis writing process. Due to time constraints, this should be investigated further in future work to reliably identify the underlying causes. One plausible explanation is that the shuffling impairs cache and transaction efficiency of the temporary storage such that I/O increases significantly.

These results suggest that `gtfs2rdf` is very efficient for large optimised real-world feeds, but significantly slower for challenging feeds. While the generalisability of these findings needs to be explored further, a runtime of 31.2 min on a 5.9 GB stress-test feed may still be considered acceptable for many use cases.

Comparability

To relate this work to conversion approaches outlined in Section 2.4, it should be noted that comparison is possible only to a limited extent. Some mappings cannot be realised by other converters that are supported by `gtfs2rdf`, and vice versa. Some mappings need to be expressed in technically very different ways or require pre-processing. For example, shape aggregation cannot be easily reproduced in standard RML. Still, a few experiments were conducted whose results are outlined in Table 4.4. Note that N-Triples was chosen as the output format wherever the Turtle format was not sufficiently comparable across the tools being compared. For example, `tarql` produces a very compact Turtle style, making use of many syntactic sugar features. This style is not supported by `gtfs2rdf`.

Since GTFS files are CSV files with additional constraints, a simple row-wise mapping from the Genius lyrics dataset was executed with `gtfs2rdf` and the generic CSV-to-RDF converter `Tarql` [16, 29]. This dataset was chosen due to time constraints: it has fairly few columns so that setting up the mapping is less

Tool	Task	Time [s]	RSS [MB]	Notes
<code>gtfs2rdf</code>	Genius lyrics CSV → N-Triples	103.0	170.0	Simple row-wise mapping
<code>Tarql</code>	Genius lyrics CSV → N-Triples	587.0	950.0	Same simple mapping; generic CSV-to-RDF tool
<code>Morph-KGC</code>	subset of GTFS.de-real → N-Triples	> 85.6	> 6926.8	Out-of-memory before completion
<code>gtfs-csv2rdf</code>	GTFS.de-real → Turtle	3750.0	99.4	Fixed GTFS-specific mapping; not fully correct and unmaintained

Table 4.4: Indicative comparison with related conversion tools. Results are not directly comparable across all rows, since datasets, mappings, and supported feature sets differ. In particular, GTFS-specific operations such as shape aggregation and explicit service-date materialisation could not be reproduced for other tools.

time-consuming compared to a GTFS mapping. At the same time, its substantial size (8.44 GB) allows for a meaningful comparison of the conversion performance. Our tool performed notably better, suggesting that the implementation principles of `gtfs2rdf` are beneficial beyond GTFS-specific mappings.

Morph-KGC was tested on a subset of the GTFS.de-real files but did not finish the run due to memory limitations. This suggests that large real-world feeds can become very demanding if a tool is not specialised for this type of data [22].

Finally, the baseline converter *gtfs-csv2rdf* was run on the GTFS.de-real feed. The simple row-wise mapping produced a very large output size of 45.7 GB with 850 million triples. This is mainly because every field from the large `shapes.txt` file is written explicitly as a triple, whereas our tool outputs all rows pertaining to one `shape_id` as one compact linestring geometry. While direct comparison is limited by the different output structure, *gtfs-csv2rdf*'s much lower throughput on this workload suggests lower practical efficiency than `gtfs2rdf` for this type of conversion.

It should be noted that both *tarql* and *Morph-KGC* are general-purpose engines, that is, their fields of application are much broader than what `gtfs2rdf` is optimised for. Regardless, we can conclude that our converter fills a real gap in the

conversion of the widely adopted GTFS specification to RDF.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The thesis addressed four research questions concerning scalability, mapping flexibility, multi-file semantics, and comparison with related tools.

With regard to scalability, we demonstrated that `gtfs2rdf` can process large real-world GTFS feeds with bounded memory usage and linear runtime, barring user-configured transformations and cross-file relations. This makes it a suitable tool for conversion tasks in a reasonable time frame on commodity hardware.

Mapping flexibility was achieved by means of a schema interface and a simple DSL embedded in C++ as well as the option to apply data transformations dynamically, allowing users to model the mapping to RDF in a way that fits their needs and data. This will also accommodate future developments in the GTFS specification.

A storage component accessible via the schema interface supports the modelling of cross-file relations via storage lookups. It is integrated into the streaming conversion process to preserve efficient execution and utilises a SQLite backend to achieve bounded memory usage.

Finally, we briefly compared `gtfs2rdf` to related tools and found that it supports features that are not yet expressible in related converters and would require pre-processing steps at the very least, thereby filling a gap in the landscape of GTFS-to-RDF conversion.

5.2 Future Work

While overall this work achieved the goals it set out to accomplish, there are still several things to be done in the future to boost performance, usability, and empirical evaluation. We will lay out some of the most promising improvements in the following table, ordered from high to low priority.

Item	Potential improvement	Possible realisation	Effort
Decouple mapping from build phase	Ease of use and flexibility	Parsing layer built on top of existing schema machinery that parses mapping files from a user-configurable directory and builds <code>Schema</code> objects accordingly at runtime	5 days
On-the-fly output compression	Reduces required disk space by a lot since RDF data compresses very well	Wrap the buffered output stream in a streaming gzip encoder or similar; refactor writer module while at it	4 days
Extend evaluation	Generalisability of results	Run benchmarks on other machines with lower / higher per-core performance; evaluate unoptimised / "messy" GTFS feeds in particular	5 days
Extend testing suite	Trust in correctness / conversion quality	Add unit tests and end-to-end tests that provoke errors; the former might require code refactoring	6 days
Extend transform library	Ease of use and feed validation during conversion	Add type checking and normalisation functions for all datatypes allowed in GTFS	3 days
Optimise storage component	Performance	Table name strings are concatenated on every storage operation → introduce caching	2 days

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor, Dr Patrick Brosi, for his valuable guidance and insightful feedback throughout the course of this thesis, as well as for introducing me to this fascinating rabbit hole that is the transit-data corner of the internet.

I would also like to thank the examiner of this work, Professor Hannah Bast, for agreeing to examine my thesis.

Last but not least, I am grateful to Alexander Richter, Till Steinmann, and Andreas Reichenbach for authoring and making available this L^AT_EX template, enabling access to a professionally typeset thesis template and thus improving the overall quality of this thesis.

Bibliography

- [1] MobilityData. ‘General transit feed specification (gtfs) schedule reference.’ (28th Oct. 2025), [Online]. Available: <https://gtfs.org/documentation/schedule/reference/> (visited on 16/02/2026).
- [2] M. Haklay and P. Weber, ‘Openstreetmap: User-generated street maps,’ *IEEE Pervasive Computing*, vol. 7, no. 4, pp. 12–18, 2008. DOI: 10.1109/MPRV.2008.80.
- [3] D. Vrandečić and M. Krötzsch, ‘Wikidata: A free collaborative knowledge-base,’ *Commun. ACM*, vol. 57, no. 10, pp. 78–85, Sep. 2014, ISSN: 0001-0782. DOI: 10.1145/2629489. [Online]. Available: <https://doi.org/10.1145/2629489>.
- [4] R. Cyganiak, D. Wood and M. Lanthaler, ‘Rdf 1.1 concepts and abstract syntax,’ W3C, W3C Recommendation, Feb. 2014. [Online]. Available: <https://www.w3.org/TR/rdf11-concepts/>.
- [5] E. Prud’hommeaux, S. Harris and A. Seaborne, ‘Sparql 1.1 query language,’ W3C, W3C Recommendation, Mar. 2013. [Online]. Available: <https://www.w3.org/TR/sparql11-query/>.
- [6] H. Bast and B. Buchhold, ‘Qlever: A query engine for efficient sparql+text search,’ in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ser. CIKM ’17, Singapore, Singapore: Association for Computing Machinery, 2017, pp. 647–656, ISBN: 9781450349185. DOI: 10.1145/3132847.3132921. [Online]. Available: <https://doi.org/10.1145/3132847.3132921>.
- [7] Ontotext. ‘Graphdb.’ (), [Online]. Available: <https://www.ontotext.com/products/graphdb/> (visited on 16/02/2026).

-
- [8] Open Geospatial Consortium, ‘Ogc geosparql — a geographic query language for rdf data,’ OGC Standard GeoSPARQL 1.1. [Online]. Available: <https://docs.ogc.org/is/22-047r1/22-047r1.html> (visited on 17/02/2026).
- [9] ISO/IEC, ‘Information technology — database languages — sql multimedia and application packages — part 3: Spatial,’ ISO/IEC Standard 13249-3:2016, 2016. [Online]. Available: <https://www.iso.org/standard/60343.html> (visited on 17/02/2026).
- [10] Algorithms and Data Structures Group, University of Freiburg, *Pfaedle, Precise map-matching for public transit feeds (generates gtfs shapes from openstreetmap data)*. [Online]. Available: <https://github.com/ad-freiburg/pfaedle> (visited on 17/02/2026).
- [11] Algorithms and Data Structures Group, University of Freiburg, *Qlever-petrimaps, Interactive map visualization of sparql query results with geospatial information*. [Online]. Available: <https://github.com/ad-freiburg/qlever-petrimaps> (visited on 17/02/2026).
- [12] Strætó bs. ‘Strætó bs: Public transport in the capital area of iceland.’ (), [Online]. Available: <https://straeto.is/en/> (visited on 17/02/2026).
- [13] OpenTransport, *Gtfs-csv2rdf, Mapping script which transforms gtfs csv into gtfs rdf*. [Online]. Available: <https://github.com/OpenTransport/gtfs-csv2rdf> (visited on 17/02/2026).
- [14] OpenTransport. ‘Opentransport, Github organization.’ (), [Online]. Available: <https://github.com/OpenTransport> (visited on 17/02/2026).
- [15] M. Kolchin. ‘A practical review of non-rdf to rdf converters.’ Published in datafabric on Medium. (8th Jun. 2018), [Online]. Available: <https://medium.com/datafabric/a-practical-review-of-non-rdf-to-rdf-converters-51686338927f> (visited on 17/02/2026).
- [16] Tarql contributors, *Tarql, Sparql for tables*. [Online]. Available: <https://github.com/tarql/tarql> (visited on 17/02/2026).
- [17] ‘Moin — mobility index.’ (), [Online]. Available: <https://mobilityindex.net/> (visited on 18/02/2026).
- [18] MoIn Project, *Gtfs2rdf, Convert gtfs data to rdf*. [Online]. Available: <https://github.com/moin-project/GTFS2RDF> (visited on 18/02/2026).

-
- [19] RMLio, *Rmlmapper-java*, *Rml mapper*. [Online]. Available: <https://github.com/RMLio/rmlmapper-java> (visited on 17/02/2026).
- [20] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens and R. Van de Walle, ‘Rml: A generic language for integrated rdf mappings of heterogeneous data,’ in *Proceedings of the 7th Workshop on Linked Data on the Web (LDOW 2014)*, ser. CEUR Workshop Proceedings, vol. 1184, 2014. [Online]. Available: https://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf.
- [21] RMLio, *Rmlstreamer*. [Online]. Available: <https://github.com/RMLio/RMLStreamer> (visited on 17/02/2026).
- [22] Morph-KGC contributors, *Morph-kgc*, *Knowledge graph materialization engine*. [Online]. Available: <https://github.com/morph-kgc/morph-kgc> (visited on 17/02/2026).
- [23] E. de Vleeschauwer, P. Maria, B. De Meester and P. Colpaert, ‘Rml-view-to-csv: A proof-of-concept implementation for rml logical views,’ in *Proceedings of the 5th International Workshop on Knowledge Graph Construction (KGCW 2024)*, co-located with the 21st Extended Semantic Web Conference (ESWC 2024), ser. CEUR Workshop Proceedings, vol. 3718, 2024. [Online]. Available: <https://ceur-ws.org/Vol-3718/paper2.pdf>.
- [24] D. Chaves-Fraga, F. Priyatna, A. Cimmino, J. Toledo, E. Ruckhaus and O. Corcho, ‘Gtfs-madrid-bench: A benchmark for virtual knowledge graph access in the transport domain,’ *Journal of Web Semantics*, vol. 65, p. 100 596, 2020, ISSN: 1570-8268. DOI: 10.1016/j.websem.2020.100596. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1570826820300354>.
- [25] ‘Opengis implementation standard for geographic information — simple feature access — part 1: Common architecture,’ Open Geospatial Consortium, OpenGIS Implementation Standard OGC 06-103r4, version 1.2.1, May 2011, Defines the Well-known Text (WKT) representation for geometries, including LINESTRING. [Online]. Available: https://portal.opengeospatial.org/files/?artifact_id=25355.
- [26] SQLite Developers, *SQLite*, *Amalgamation source code (sqlite-amalgamation-3510200.zip)*, version 3.51.2, Includes `sqlite3.c/sqlite3.h`. Public domain

- dedication., 9th Jan. 2026. [Online]. Available: <https://www.sqlite.org/download.html> (visited on 03/03/2026).
- [27] A. B. Kahn, ‘Topological sorting of large networks,’ *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962. DOI: 10.1145/368996.369025. [Online]. Available: <https://doi.org/10.1145/368996.369025>.
- [28] OEG-UPM, *Gtfs-madrid-bench script repository*, version v1.3.2, Benchmark data generation script, 2025. [Online]. Available: <https://github.com/oeg-upm/gtfs-bench> (visited on 10/03/2025).
- [29] carlogdcj. ‘Genius song lyrics with language information.’ Kaggle dataset. (), [Online]. Available: <https://www.kaggle.com/datasets/carlogdcj/genius-song-lyrics-with-language-information> (visited on 11/03/2026).

Appendix

1 Command-line interface

Listing 5.1: Command-line help output of `gtfs2rdf`.

```
Gtfs->Rdf converter
Note: garbage in, garbage out. Validate your Gtfs feed first.
A pre-run with stats enabled is recommended to validate schemas and
estimate requirements before conversion.

Examples:
  gtfs2rdf feed.zip --format nt
  gtfs2rdf --feed feed.zip --output out/

Usage:
  gtfs2rdf [options] Gtfs_ZIP

Standard options:
  -o, --output arg  Output directory or 'stdout' (default: .)
  -p, --pre-run     Validate schemas and perform a dry run: with stats
                   enabled, print file headers and roughly estimate
                   output size and peak RAM usage without writing output.
                   Recommended with --spec-dump to inspect conversion
                   details before conversion.
  --format arg     Output format: ttl|nt (default: ttl)
  --overwrite      Overwrite existing output files.
  -h, --help       Show help

RAM / runtime options:
  --read-buffer-size arg  Read buffer size in MB (bigger = more RAM,
                          fewer reads) (default: 80.000000)
  --write-buffer-size arg Write buffer size in MB (bigger = more
                          RAM, fewer flushes) (default: 80.000000)
  --storage-buffer-size arg
```

```

                                Size in MB of RAM that may be allocated
                                for persistent storage cache between Gtfs
                                files (bigger = more RAM, might be faster)
                                (default: 500.000000)
--tmp-dir arg                    Temporary directory for intermediate files
                                (default: .)

```

Diagnostics / advanced options:

```

--spec-dump                      Dump mapping spec to disk
--warning-level arg             Warning verbosity: quiet|warning|debug
                                (default: quiet)
--stats arg                    Statistics output: quiet (none), brief
                                (overview), verbose (overview per-file)
                                (default: quiet)

```

2 Mapping DSL Reference

This appendix documents the placeholder DSL used in **schema user statements**. A user statement is either (i) a triple-emitting mapping rule in `TRIPLES`, or (ii) a storage-only user statement in `NO_WRITE_INSTRUCTIONS`. During compilation, each user statement is transformed into an executable object of class `Instruction`. Thus, the DSL is written declaratively by the user at schema definition time and executed later in compiled form.

Within a string-valued part of a user statement, any substring of the form `{ ... }` is called a **placeholder**. A placeholder is evaluated once per GTFS row.

2.1 Placeholder basics

Whitespace. Whitespace is ignored everywhere except inside quoted literals `"..."`.

Empty output. If a placeholder evaluates to the empty string, the entire compiled instruction is skipped. Hence, no triple is emitted and no later placeholders of that instruction are processed.

Execution granularity. A compiled instruction is evaluated once per input row of the corresponding GTFS file.

2.2 Placeholder forms (summary)

Table 5.1 summarises the supported placeholder forms.

Table 5.1: Summary of placeholder forms in the mapping DSL.

Form	Meaning
<code>{col}</code>	Read column <code>col</code> from the current GTFS row.
<code>{"literal"}</code>	Use a literal string value. Inside surrounding C++ strings, the quotes must be escaped.
<code>{arg_1,...,arg_N trf_1 ... trf_M}</code>	Apply a transform chain from left to right. The first transform sees all arguments; each later transform sees exactly one pipeline value.
<code>{VAR@file.txt}</code>	Read variable <code>VAR</code> from storage context <code>file.txt</code> . Missing reads yield the empty string.
<code>{expr > VAR@file.txt}</code>	Write the computed output of <code>expr</code> to variable <code>VAR</code> in context <code>file.txt</code> .
<code>{k_1,...,k_r : v_1,...,v_s > map@file.txt}</code>	Keyed raw store. If $s = 1$, this writes to a <code>MULTI_MAP</code> ; otherwise, it writes a tuple to a <code>TUPLE_MAP</code> .
<code>{k_1,...,k_r : v_1,...,v_s trf_1 ...> map@file.txt}</code>	Keyed computed store. The transform chain is evaluated first, and the final output is then stored under the given key.
<code>{k : (raw_1,...,raw_s), extra_1,...,extra_t pred_1 ...> map@file.txt}</code>	Filter-store-raw. The transforms act as a predicate; if the computed result is non-empty, the raw tuple in parentheses is stored.

2.3 Arguments

Arguments are the comma-separated items before the first `|`. They may be GTFS column references, quoted literals, or variable reads.

Table 5.2: Argument kinds in placeholders.

Kind	Example	Meaning
Column	<code>stop_id</code>	Value of CSV column <code>stop_id</code> in the current row.
Literal	<code>"de-CH"</code>	Literal string value. Whitespace inside the quotes is preserved.
Variable read	<code>FEED_LANG@ feed_info.txt</code>	Read variable <code>FEED_LANG</code> from storage context <code>feed_info.txt</code> .

2.4 Transforms and chaining semantics

A placeholder may contain a transform pipeline:

```
{ arg1, arg2, ..., argN | transform1 | transform2 | ... | transformM }
```

Dataflow rule. Let the initial argument list be `ARGS = [arg1, ..., argN]`.

- The first transform receives the full list `ARGS`.
- Each subsequent transform receives exactly one input value, namely the output of its predecessor.

Transform2Many. A `Transform2Many` produces multiple output strings. This fans out the compiled instruction: the instruction is emitted once per non-empty produced value. Any later `Transform2One` is applied element-wise to those produced values.

Restriction. Within one compiled instruction, at most one `Transform2Many` may occur. If present, it determines the fan-out of that instruction.

2.5 Storage writes and keyed writes

Storage writes are side effects of placeholder evaluation and use the syntax

```
{ ... > STORE_NAME@context_file.txt }
```

The context must end in `.txt`.

Keyed writes. A keyed write uses `:` to separate keys from values:

```
{ key1, key2 : value1, value2 > store_name@context_file.txt }
```

Storage kinds. The storage backend supports three abstractions:

- **VARIABLE:** a single string value
- **MULTI_MAP:** key \mapsto many strings
- **TUPLE_MAP:** key \mapsto many tuples

Operational modes.

- **STORE_RAW:** keyed write without transforms; the right-hand side is stored directly
- **STORE_COMPUTED:** keyed write with transforms; the final computed output is stored
- **FILTER_STORE_RAW:** the transform chain acts as a predicate; if non-empty, the raw tuple in parentheses is stored

2.6 Contexts and dependency visibility

Contexts serve two roles:

- **Namespace separation:** storage entries are scoped by context.
- **Dependency planning:** explicit `@file.txt` annotations expose cross-file dependencies so that a safe processing order can be computed.

Rule of thumb. Variable reads should always be written as `VAR@other_file.txt`. Likewise, when a transform reads from storage, the transform call should be annotated as `transform@other_file.txt` so that the dependency is visible at planning time.

2.7 Storage-only user statements

Storage-only user statements are evaluated before the triple-emitting user statements of the same schema. They are used to initialise variables, multi-maps, or tuple-maps without directly emitting RDF output.

2.8 Canonical examples

Listing 5.2 gives a compact set of examples.

Listing 5.2: Canonical examples of the mapping DSL.

```
# 1) Column substitution
{ stop_id }

# 2) Literal
{ "de" }

# 3) Transform chain
{ a, b | exJoinAB | exToUpperAscii }

# 4) Variable read
{ FEED_LANG@feed_info.txt }

# 5) Variable write
{ feed_lang | exToUpperAscii > FEED_LANG@feed_info.txt }

# 6) Keyed raw store (tuple-map): shape_id -> (seq, lon, lat)*
{ shape_id : shape_pt_sequence, shape_pt_lon, shape_pt_lat > ←
  shapes@shapes.txt }

# 7) Keyed computed store (multi-map): service_id -> disabled_date*
{ service_id : date, exception_type | isDisabledDate | ←
  convertDate2xs_unchecked
  > disabled_dates@calendar_dates.txt }

# 8) Fan-out via Transform2Many
{ tags | exSplitSemicolon | exToUpperAscii }
```

2.9 Implementation limits

For predictability, the parser enforces conservative default limits:

- Maximum number of arguments per placeholder: `MAX_ARGS`
- Maximum number of transforms per placeholder: `MAX_TRANSFORMS`
- At most one `Transform2Many` per compiled instruction

These limits can be adjusted at build time if required.

3 GTFS Reference Graph

Nodes are GTFS files. An arrow $A \rightarrow B$ means file A contains field(s) that reference identifiers in file B ; edge labels name those fields. Solid arrows indicate direct ID references. Dashed arrows indicate conditional/meta references (e.g. `calendar_dates.txt` linking to `calendar.txt` when both share `service_id`, and `translations.txt` requiring `feed_info.txt` / potentially targeting many tables). Non-reference constraints (uniqueness, mutual exclusivity, etc.) are not shown.

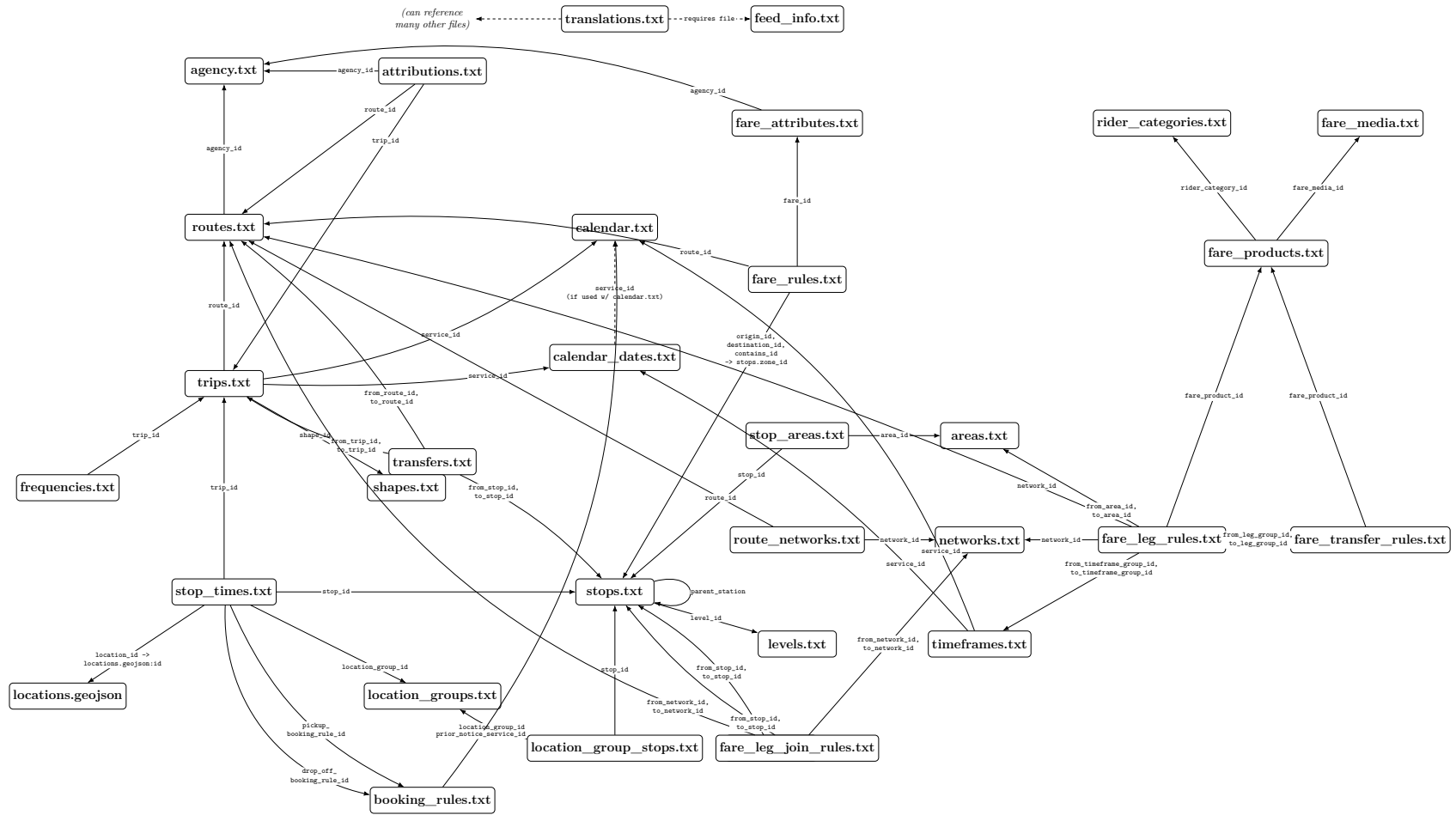


Figure 5.1: GTFS Schedule file reference graph (A→B means A references B) as per [1].

