Report on Updated ASA System Architecture

Deliverable ID:	**D2.21**
Project acronym:	**AWARE**
Grant:	**101167442**
Call:	**HORIZON-SESAR-2023-DES-ER-02**
Topic:	**HORIZON-SESAR-2023-DES-ER2-WA2-4**
Consortium coordinator:	**FTTS**
Edition date:	**11 July 2025**
Edition:	**01.00**
Status:	**Official**
Classification:	**PU**

Abstract

This document investigates improved options for the architecture of the Artificial Situational Awareness (ASA) system, a core component of the AWARE project, which aims to develop artificial intelligence (AI) support for air traffic controllers, where the AI component has information about task-relevant context and the controller's attention. Building on prior work in the AISA project, this document explores replacing the original Prolog-based knowledge graph (KG) implementation in the ASA system with either a native RDF-based or a relational implementation, evaluating both in terms of performance, rule expressiveness, and development effort. The document also introduces a cloud-native lakehouse architecture for scalable ingestion and indexing of heterogeneous, high-velocity data sources. This KG Lakehouse allows for on-demand KG generation from the indexed data and is compatible with both a native RDF-based implementation and a relational implementation of KG management for the downstream data processing tasks.

Co-funded by the European Union

Authoring & approval

Author(s) of the document

Organisation name	Date
JKU	30 June 2025

Reviewed by

Organisation name	Date
FTTS	30 June 2025

Approved for submission to the SESAR 3 JU by

Organisation name	Date
FTTS	11 July 2025
JKU	11 July 2025
LFV	11 July 2025
SLOT	11 July 2025
TERN	11 July 2025
UkSATSE	11 July 2025
UPM	11 July 2025
IFATCA	11 July 2025
ZHAW	11 July 2025

Rejected by

Organisation name	Date

Document history

Edition	Date	Status	Organisation author	Justification
00.01	02 June 2025	Draft	JKU	Initial draft
00.02	27 June 2025	Draft	JKU	Draft for review
00.03	30 June 2025	Draft	JKU	Final draft
01.00	11 July 2025	Official	JKU	Submitted version

Copyright statement

Disclaimer

AWARE

ACHIEVING HUMAN-MACHINE COLLABORATION WITH ARTIFICIAL SITUATIONAL AWARENESS

AWARE

This document is part of a project that has received funding from the SESAR 3 Joint Undertaking under grant agreement No 101167442 under European Union's Horizon Europe research and innovation programme.

Table of contents

List of figures

List of tables

Co-funded by the European Union

1 Executive summary

The AWARE project's main objective is the development of artificial intelligence (AI) support for air traffic controllers (ATCOs), where the developed AI components are "aware" of an ATCO's focus and the tasks at hand. At the core of the developed solution is an artificial situational awareness (ASA) system, which integrates information from various sources, including messages conforming to various ATM exchange models but also attention-tracking hardware and software, in a knowledge graph (KG) to perform rule-based reasoning and predictive analytics over the KG.

The KG of the ASA system serves as an integrated source of information for various predictive-analytics applications but also rule-based reasoning to derive new contents from the existing facts in the KG. Regarding the specification of rules for deriving new contents from the existing facts in the KG of the ASA system, the following main options are possible.

- **Prolog** or **Datalog**. Logical formulae in the form of Horn clauses, which essentially correspond to if-then rules, can serve for the declarative definition of rules for deriving facts from an existing KG. Datalog is a simplified version of Prolog, geared towards database management and querying.

- **SPARQL-based and/or SHACL-based rules**. The SPARQL Protocol and RDF Query Language [29] can be used to construct new facts from a given KG by specifying basic graph patterns. For a KG to be queried using SPARQL, the KG must be stored using the Resource Description Framework (RDF), which represents the KG's facts in form of triples of subject, predicate, and object. The Shapes Constraint Language [111], which primarily aims at providing a language for describing constraints over RDF data, provides a mechanism for the definition of SPARQL-based rules, where the rules are defined as SPARQL CONSTRUCT queries.

- **SQL**. While not a rule language for KGs *per se*, SQL-based view definitions can be used to emulate rule-based derivation of facts in KGs. The tables in the database correspond to the base facts of the KG, whereas (possibly materialized) views, specified using SQL queries, correspond to derived facts in the KG.

In the AISA project, the results of which inform the development activities of the AWARE project, an RDF database stored the KG, whereas Prolog served for the definition of complex logical rules, which required a *Prolog Mapper* to export Prolog facts from the KG stored in the RDF database. The Prolog facts were then used as input for an external Prolog engine executing the Prolog rules. The results of the rule evaluation would then have to be imported back into the KG. This solution had a considerable performance overhead. In addition, the overall system became less understandable while being more difficult to debug and to optimize.

Therefore, in this document, we evaluate two implementation options for the ASA system both in terms of run time performance and other required effort for management and development.

- **Native RDF-based implementation**. An RDF database, e.g., Fuseki, GraphDB, or RDFox, stores the facts of a KG in form of triples of subject, predicate and object. Those triples may be grouped into named graphs to allow for contextualization of the represented knowledge facts. For querying the KG, RDF databases typically support both Datalog and SPARQL. SHACL rules

Co-funded by the European Union

can be evaluated over the contents of an RDF database, typically by using external libraries. The main advantage of the RDF-based implementation is the flexibility of the KG schema.

- **Relational implementation**. When using a relational database management system, e.g., PostgreSQL, for the representation of a KG, the schema of the KG is less flexible and may require additional development effort. Relational database management software, however, is a tried-and-tested technology and readily available for production use.

Since SPARQL-based rules and/or SHACL-based rules (which are essentially SPARQL-based rules) cannot adequately handle complex, mutually dependent and/or recursive rules, we do not further consider this option as viable for the ASA system at the moment. While the trial runs in the AWARE project may not require the formulation of complex rules, future applications and production systems may well formulate more complex rules. In this case, new facts must be iteratively derived by repeatedly evaluating the defined set of rules until the KG reaches a fixed point where no new facts are added. In this case, a stratification mechanism must be implemented to allow for the iterative processing of complex, mutually dependent and/or recursive rules. Nevertheless, since RDF databases typically support SPARQL, the option of using SPARQL-based rules will still be open to a native RDF-based implementation in the future, even though the formulation of such rules will require the implementation of a stratification mechanism.

Using Datalog in conjunction with a native RDF-based implementation of the ASA system could serve to represent complex, mutually dependent and/or recursive rules, with the system being able to adequately handle such rules. A relational implementation would allow for representation of iterative, mutually dependent and/or recursive rules to a certain degree, using the (recursive) WITH statement of SQL in the definition of views representing rules. However, more complex rules will likely require additional stratification mechanisms that a native RDF-based implementation already provides out of the box for the evaluation of Datalog-based rules.

Due to the restrictive nature of the license agreement of the commercial, native RDF-based implementation used in experimental development, this document does not publish performance data of the native RDF-based implementation. In any case, a direct comparison of the performance of the native RDF-based implementation and the relational implementation cannot be used as the sole basis for design decisions.

In general, the relational implementation presented in this document trades flexibility for performance by relying on a fixed schema for the KG and by offering the possibility to implement an optimized set of SQL-based and externally represented (possibly Java-based) rules on a comparatively low level. However, any potential performance gain through this type of optimization comes at the expense of flexibility, both in terms of what type of information can be represented in the KG and in terms of queries that can be formulated.

This document also introduces a cloud-native lakehouse architecture for efficiently ingesting and indexing high volumes of a variety of raw data files arriving at high velocity, which is a prerequisite for deployment of the system in a production setting. From the indexed raw data files, a KG tailored to the requirements of a specific application and task can be generated on demand. The proposed KG Lakehouse is compatible with the use of both the native RDF-based implementation and the relational implementation in downstream data processing.

2 Introduction

2.1 Purpose of the document

This document revises the architecture of the ASA system as it was developed previously in the AISA project. In particular, this document investigates the switching from Prolog to either a native RDF-based implementation or a relational implementation for knowledge graph (KG) management, depending on both performance and other considerations. In addition, this document investigates the design and implementation of a cloud-native lakehouse architecture, which can serve as the basis for deployment of the ASA system in a production setting.

2.2 Intended readership

This document serves as the basis for design decisions during the development of the ASA system in the AWARE project. System developers building on the results of the AWARE project may refer to this document to obtain guidance regarding the implementation of production systems. Beyond the ATM domain, researchers and practitioners may use the insights to inform the development of KG management systems in time-critical settings.

2.3 Background

The AWARE project builds on the results of the AISA (AI Situational Awareness Foundation for Advancing Automation) project, which has received funding from the SESAR Joint Undertaking under grant agreement No 892618, under European Union's Horizon 2020 Research and Innovation programme.

2.4 Structure of the document

This document Is organized as follows. Chapter 3 presents the relevant state of the art. Chapter 4 describes the benchmarking setup for evaluating the performance of the presented design alternatives. Chapter 5 describes a native RDF-based implementation for knowledge graph management in the ASA system. Chapter 6 describes mapping Datalog-based rules to SQL for a relational representation of KGs. Chapter 7 describes potential relational storage layouts for RDF data. Chapter 8 further discusses variants of relational representation of rules. Chapter 9 investigates the handling of dynamic data in a relational KG implementation. Chapter 10 presents the results of performance experiments with the relational system. Chapter 11 compares the native RDF-based KG implementation with a relational implementation. Chapter 12 describes the necessary steps for migrating from a native RDF-based KG implementation to a relational implementation. Chapter 13 presents a cloud-native lakehouse architecture for dealing with a large volume of a variety of data arriving at high velocity in KG management.

2.5 Glossary of terms

Term	Definition	Source of the definition
Knowledge graph	*A knowledge graph represents real-world entities and their relationships. Most of the knowledge graph's contents are instance-level facts, although knowledge graphs may also include terminological or ontological knowledge, representing a kind of vocabulary for the knowledge graph.*	*Schuetz et al. [25]*

Table 1: Glossary of terms

2.6 List of acronyms

Term	Definition
AI	*Artificial intelligence*
AISA	*AI Situational Awareness Foundation for Advancing Automation)*
ASA	*Artificial situational awareness*
ATCO	*Air traffic controller*
AWARE	*Achieving Human-Machine Collaboration with Artificial Situational Awareness*
EDB	*Extensional database*
IDB	*Intensional database*
KG	*Knowledge graph*
RDF	*Resource Description Framework*
SHACL	*Shapes Constraint Language*
SPARQL	*SPARQL Protocol and RDF Query Language*
SQL	*Structured Query Language*

Table 2: List of acronyms

3 State of the art

This chapter provides a brief introduction to the concept of contextualized KGs. Furthermore, this chapter reviews related work on management of KGs using both a native RDF-based environment and a relational setting. An introduction to the basics of benchmarking as relevant for the purposes of the experiments in this document concludes this chapter.

3.1 Contextualized knowledge graphs

A major advantage of using graphs is flexibility when conceptualizing, representing, and integrating diverse and incomplete data [5]. Using graphs instead of relational models or NoSQL approaches may be beneficial for domains in which edges and paths capture complex and different relations between entities [5], [6]. In addition to standard relational operators like joins and unions, graph query languages may also support finding entities along paths of arbitrary length [5], [7].

Figure 1 shows a directed edge labelled graph that describes a flight, its destination, and its origin. The idea behind this graph is based on Hogan et al. [5]. The facts in the graph are essentially true in terms of a specific context [5]. For example, in terms of a temporal context [8], [9], [10], [11] the city of Santiago has existed as a city since 1541 and flights from Arica to Santiago first occurred in 1956 [5]. For a time before 1541, the fact of Santiago being a city would be wrong. However, in Figure 1 this information is not captured. A similar context dependency can often be observed with respect to provenance [12], [13], [14]. For example, the data about flight *f1* might be retrieved from a website that tracks flights. Therefore, data about *f1* is true with respect to this website at a specific time point [5]. Multiple forms of context may be used and combined, as, for example, geographic, temporal, and provenance [5]. When indicating that Arica is a Chilean city (geographic) since 1883 (temporal) as defined in the Treaty of Ancón (provenance), a combination of contexts is used [5]. Context can therefore be referred to as the scope of truth, which gives insight into the boundaries for which given data is held as true [15], [16]. Figure 1 holds context as implicit [5]. Making context explicit allows for an interpretation of data from different perspectives [5]. For example, explicit temporal context for the graph shown in Figure 1 would allow a retrieval of data that was held true some hours ago. It would then be possible to recognize last-minute delays in flight schedules. Explicit provenance context would allow the exclusion of data from a website that is now considered dubious [5]. For example, if the information about flight *f1* was collected from a website that is prone to show inaccurate data, an application may apply additional checks before processing the data further.

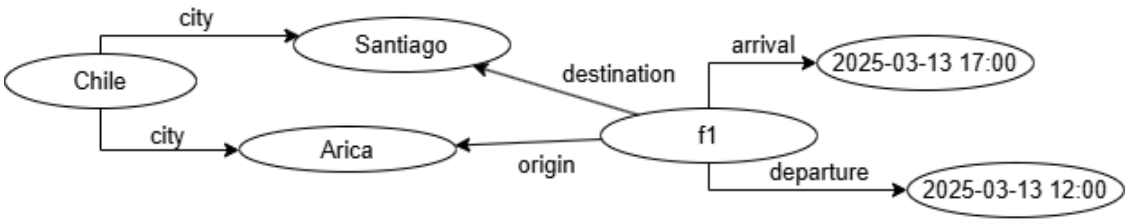


Figure 1: Directed edge labeled graph

Explicit context representation may be achieved via different approaches [5]. Hogan et al. describe direct representation, reification, higher-arity representation, annotations, and other contextual frameworks as possible approaches [5]. The different approaches to representing context are briefly shown in this section and visualized by using a fictional flight between two cities.

The most straightforward way to achieve explicit context representation is by using direct representation. For this approach, context data is represented in the same way as the rest of the data in the graph [5]. Figure 2 extends the previously shown directed graph in Figure 1 by introducing the entities *valid from* and *valid until*. The entity *f1* represents its temporal context directly via the two associated *valid from* and *valid until* edges [5], [10]. Generally, for this direct approach, the exact way of representing context may differ from application to application. However, to represent context in a more standardized way, alternatively the Time Ontology [8] or the PROV Data Model [13] may be used to represent temporal or provenance context respectively [5].

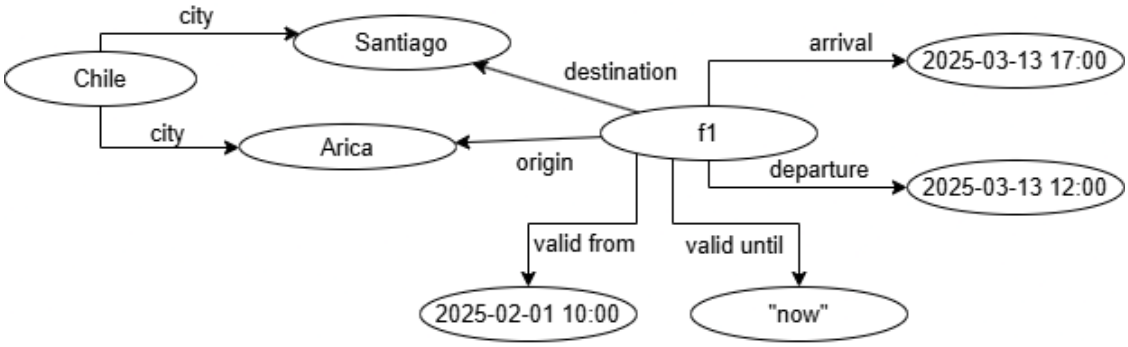


Figure 2: Extended directed edge labeled graph with explicit context representation. The edges valid from and valid until show an explicit temporal context, realized with direct context representation.

Unlike with direct context representation, reification may be used to define the context of edges instead of entities [5]. When using reification, the edge is seen as a separate entity with its own associated context description [5]. For example, if an underway flight *f1* must make a detour due to weather conditions, the validity of edges changes in a temporal context. Consider flight *f1* in Figure 2 is delayed slightly, and therefore the arrival time changes. As the rest of the shown graph remains unaffected, concerning context representation, ideally only the temporal context of the edge *f1 - arrival - 2025-03-13 17:00* would change. Figure 3 shows three different approaches to reification: RDF reification [5], [17], n-ary relations [5], [17] and singleton properties [5], [18]. Which of those approaches is used depends on the specific use case and the system in practice [5]. Generally, reification does not assert the associated edge. However, it is possible to reify an edge to denote its end of validity [5].

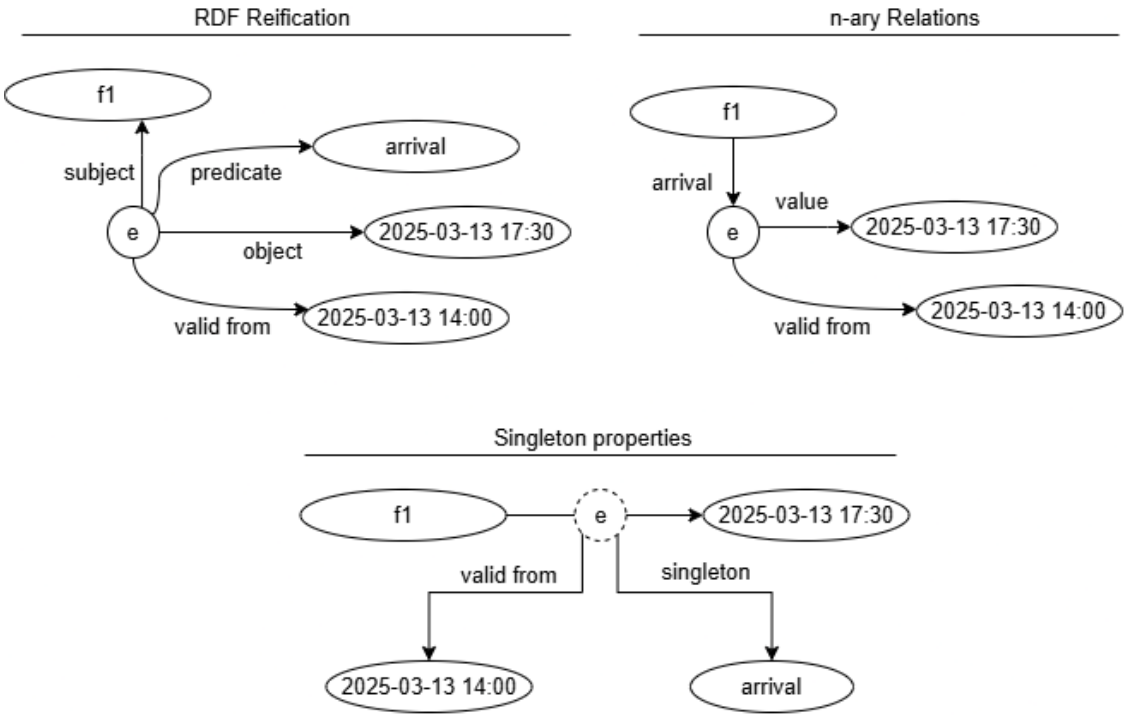


Figure 3: Three different reification approaches for temporal context representation.

Another approach to encode context is to extend the graph model [5]. Hogan et al. [5] differentiate between using named graphs, property graphs, and using the RDF-extension RDF*. Figure 4 shows a comparison between those three approaches. Choosing an appropriate variant is again dependent on the specific use case, but in general, using named graphs is considered the most flexible one, whereas RDF* is the least [5]. Using named graphs is also the approach that was chosen by the preceding AISA project for context representation [19]. Named graphs are also used for context representation in the RDF-based approach for this work, developed by Schwarz [1].

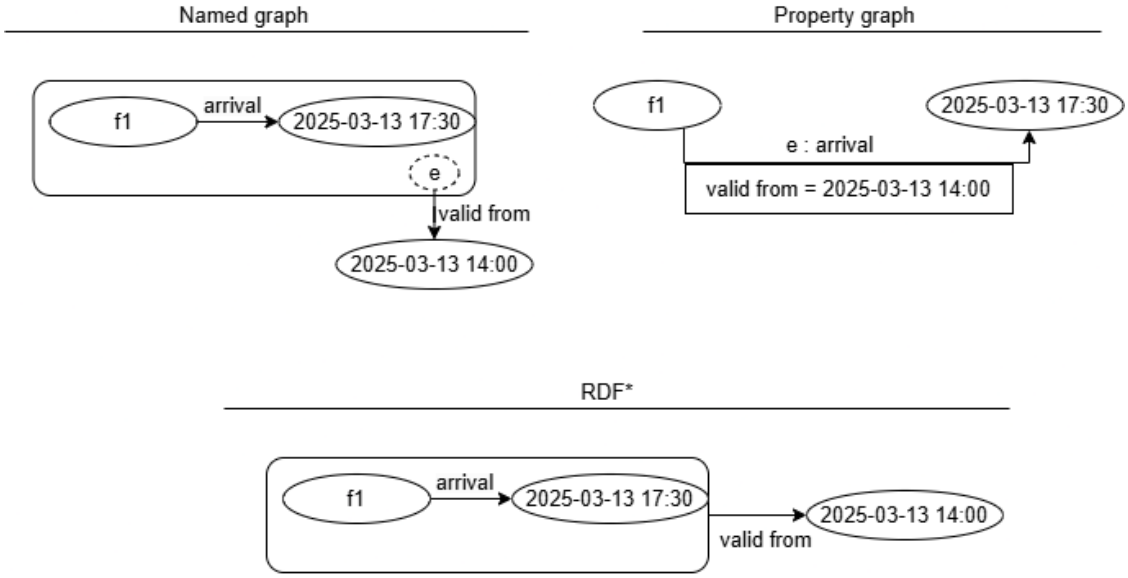


Figure 4: Three different approaches of higher-arity context representation on an edge.

Previous approaches covered in this section focused on the aspect of representing context. With *annotations* it is possible to enable automated context-aware data processing [5]. There are annotation approaches that model specific context domains, as, for example, the *Temporal RDF* for a time context or *Fuzzy RDF* which allows the annotation of edges with a degree of uncertainty [5]. However, there are also additional domain-independent frameworks such as *Annotated RDF* [20], [21], [22] which supports a representation of various contexts [5].

Finally, there are other additional frameworks for modelling and reasoning about contexts in graphs [5]. One of those frameworks concerns the use of *contextual knowledge repositories* (CKRs) [23]. CRKs allow the assignment of subgraphs to partially ordered dimensions, which then allow a selection of subgraphs at different context granularity [5]. Schuetz et al. [24], [25] built on those CKRs and proposed a contextualization of KGs based on the multidimensional and hierarchical structure of the online analytical processing (OLAP) cube model [25].

3.2 Knowledge graph management

This section covers approaches to storing and managing KGs in different technologies. For a native approach, RDF-based technologies associated with the semantic-web stack are described. For a relational model, *vertical stores*, *horizontal stores* and *property stores* are covered as storage variants. Additionally, from a management or usage point of view, different ways of exposing data to consumers who require data in an RDF-based layout are discussed. This includes utilization of a *relational to RDF mapping language (R2RML)* and *RDF views*.

3.2.1 Native RDF-based

This section gives a brief overview of native RDF-based technologies of the semantic-web stack that were used in AWARE. This includes basics of the Resource Description Framework (RDF), the Turtle format, and the query language SPARQL.

The Resource Description Framework (RDF) is a framework developed by the World Wide Web Consortium (W3C) for describing information in the web [17]. It enables the representation of information in the form of triples, consisting of a subject, predicate, and object, thereby allowing the explicit definition of relationships between data [17]. URIs (Unique Resource Identifiers) are used to identify the entities and the predicates connecting them [17]. These triples collectively form an RDF graph, which is structured as a directed graph and facilitates the modeling of networks of information [17]. RDF serves as the foundation of the Semantic Web by providing a standardized method for structuring and exchanging data. A *named graph* in RDF extends the classical RDF model by allowing a set of triples to be grouped under a unique identifier [17]. While a conventional RDF graph consists of a set of triples without an explicit name, a named graph enables the identification and management of multiple graphs within an RDF dataset. [26] This approach facilitates versioning and contextualization of data through the corresponding graph name [5], [27]. Named graphs are used for context representation in the native-based approach that was investigated in this work. In the specific case of AWARE, named graphs allow for the representation of data sources and temporal context. The Turtle format (Terse RDF Triple Language) is a compact notation for representing RDF data. It employs a straightforward syntax to express subject-predicate-object relationships, which are used for modeling semantic data [28]. Listing 1 shows a simple example of Turtle syntax.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ex: <http://example.org/> .
ex:Person1 rdf:type ex:Human ;
   ex:hasName "Alice" ;
   ex:hasAge 25 .
```

Listing 1: Example Turtle syntax

First, the prefix *@prefix ex: <http://example.org/>*. is defined, establishing *ex:* as an abbreviation for the URI of the namespace *http://example.org/*. The resource *ex:Person1* represents an entity classified as a human through the triple *ex:Person1 rdf:type ex:Human*. This indicates that *ex:Person1* is an instance of the class *ex:Human*. Furthermore, this entity possesses two literals as properties. First, *ex:hasName "Alice"* assigns the name as a string literal, and second, *ex:hasAge* 25 specifies its age as a numeric literal.

SPARQL (SPARQL Protocol and RDF Query Language) version 1.1 is a standardized query language and protocol for RDF databases, developed by the W3C [29]. It enables the extraction and manipulation of data stored in RDF format [29]. SPARQL supports complex queries, including pattern matching, filtering, and aggregation [29]. Listing 2 shows a basic SPARQL query for retrieving data.

```
PREFIX ex: <http://example.org/>
SELECT ?name ?age
FROM NAMED <http://example.org/graph1>
WHERE {
  GRAPH <http://example.org/graph1> {
    ?person ex:name ?name .
    ?person ex:age ?age .
  }
}
```

Listing 2: Example SPARQL query

The SPARQL query in Listing 2 uses the prefix ex to provide shorter identifiers such as ex:name and ex:age. The SELECT ?name ?age clause defines the variables for output. The FROM NAMED <http://example.org/graph1> statement specifies that the data originates from the named graph http://example.org/graph1. Within the GRAPH clause, the pattern for querying this graph is defined. Given the input data presented in Listing 3, the SPARQL query in Listing 2 produces the result displayed in Table 3.

```
@prefix ex: <http://example.org/> .

<http://example.org/graph1> {
  ex:person1 ex:name "Alice" ;
          ex:age 30 .

  ex:person2 ex:name "Bob" ;
          ex:age 25 .
}
```

Listing 3: Example RDF Graph

Name	Age
Alice	30
Bob	25

Table 3: Result of the SPARQL Query

3.2.2 Relational knowledge graphs

First, *vertical stores*, *horizontal stores* and *property stores* are briefly covered as means of storing RDF data in relational databases [30], [31], [32]. Second, ways of exposing and translating existing relational data in RDF form and vice versa are discussed.

Vertical stores (or also *triple stores*) use a single relational table to store triples directly in the form of *subject-predicate-object* [32]. In this way, each RDF triple is directly transferred into a single database tuple [32]. By using the fixed *subject-predicate-object* schema of a *vertical store*, the dynamic schema

of RDF can be handled without needing to change the relational schema [32]. However, this results in scalability limitations and potentially multiple self-joins for querying the data [32]. To mitigate the impact of necessary self-joins when querying the data and to achieve efficient querying, it is necessary to use specialized techniques or architectures [32]. This can be achieved by using a proposed generic architecture for storing and querying RDF data by Broekstra et al. [33], or by using a storage strategy that builds on the idea of vertical partitioning proposed by Weiss et al. [34].

Another way to store RDF data in a single table is via *horizontal stores* [32]. A table of a *horizontal store* uses one column for each predicate and one for each subject [32]. If two given triples in the form of *(subject, predicate, object)* are equal in terms of their subject and predicate, this results in a set of values {o1, o2} for the corresponding cell in row *subject* and column *predicate* [32]. For RDF triples with different subjects and/or predicates, additional rows and columns are created, respectively [32]. Depending on the RDF data of the use case, this might result in a sparsely populated table as missing values are assigned null values [32].

To solve the issues of sparsely populated tables and sets of object values, a vertical separation of the table into *property tables* was proposed [32]. The general idea is to form multiple smaller tables that capture triples with related predicates [32]. For this approach, it is necessary to identify related predicates [32]. Identification of predicates that often occur together would minimize the number of null values in the tables while still maximizing the size of each group of predicates [35]. Finding those sets of predicates could be achieved by association rule mining [35]. Other possibilities could be a separation based on access patterns of applications if known or also the values of the predicate *rdf:type* [32].

When storing a previously RDF-based KG in a relational system, the way data is accessed might change from SPARQL queries on the KG to SQL queries on the relational system. This might not be feasible for a specific use case if conceptually data access via queries on a KG is still required. A solution to this problem might be an *Ontology-Based Data Access (OBDA)* framework. When using an *OBDA* framework, queries are not posed over the data layer directly but over a conceptual layer and then translated to the data layer [36]. The conceptual layer may be given in the form of an ontology with a defined shared vocabulary, whereas the data layer is comprised of one or more data sources [36]. In this case, RDF may be used for the conceptual layer and SPARQL as the matching query language [36]. For the data layer, relational databases might be used [36]. To bridge the gap between a relational data layer and an RDF-based conceptual layer, the W3C specified mapping standards between relational databases and RDF datasets [36], [37], [38]. In general, two different approaches to achieve a mapping between relational and RDF data were introduced: the *direct mapping* [38] and mapping via *R2RML* [37]. The direct mapping serves as a method for basic transformation from relational data to RDF, without the possibility for using custom mapping rules, specific vocabulary, or structure [37], [38]. An *R2RML*, on the other hand, allows for target vocabularies and structures of the author's choice, specifically tailored to the underlying relational schema [37].

By using relational database to RDF (RDB2RDF) mappings, relational databases can be exposed as virtual RDF graphs [36]. Those virtual graphs may be kept virtual and queried during query execution, or if necessary, they can be materialized, and the generated RDF triples may then be stored separately in an RDF triple store [36]. When materialization is not necessary, the query may be posed on the virtual RDF graphs and then automatically rewritten and translated to an SQL query [36]. This query is then delegated to the DBMS for execution [36]. Delegating query execution to the underlying DBMS might be preferable due to the maturity of an established DBMS [36]. For example, potential utilization

of their robust transaction support or security measures may prove beneficial for certain use cases [36]. The OBDA system *ontop* serves as an open source and mature system to provide, for example, SPARQL endpoints or an OWLAPI with Sesame query engine [36].

3.3 Benchmarking

Benchmarks are used to compare the performance of one system to another [39]. Traditionally, this was limited to just the amount of useful work that could be achieved while also comparing the amount of time and/or resources used in the process [39]. However, this scope for benchmarks has changed in recent years and was extended to cover other properties that might be more relevant for the specific use case than just the performance [39].

Benchmarks are described by three key aspects: *metrics*, *workloads* and *measurement methodology* [39]. *Metrics* are the values derived from the acquired measurements of the benchmark, *workloads* determine the scenario and conditions under which the measurements are produced, and the *measurement methodology* describes the end-to-end process for benchmark execution to collect measurements and benchmark results [39].

Benchmarks might also cover other system quality attributes beyond performance, as for example *functional suitability*, *performance efficiency*, *compatibility*, *usability*, *reliability*, *security*, *maintainability* and *portability* [39]. However, single benchmarks will not be able to cover each system quality attribute, potentially requiring more than one benchmark depending on which system quality attributes should be covered [39]. In terms of classical performance benchmarking, Kounev et al. [39] refer to the work of Lilja [40], where benchmarking strategies are differentiated into *fixed-work benchmarks*, *fixed-time* benchmarks, and *variable-work and variable-time* benchmarks.

Fixed-work benchmarks use the execution time as the comparison metric between systems [39]. The execution time describes the amount of time that is needed to execute a fixed number of units of work [39]. In comparison, in *Fixed-Time* benchmarks, the amount of work is variable and is used as the metric to compare different systems instead [39]. Finally, a *variable-work and variable-time* benchmark might be ideal in cases where solution quality can be improved as long as additional computation time is available, which means that by keeping both work and time variable, maximum flexibility can be achieved [39].

Benchmarks underlie several different quality criteria that have to be taken into consideration when designing a benchmark: *relevance, reproducibility, fairness, verifiability,* and *usability* [39]. *Relevance* concerns how interesting the behavior and results of the benchmark are to a given user and is considered as high priority [39]. *Relevance* is dependent on the usage scenario, as a single benchmark might be of higher interest for one user or in one specific scenario while being practically useless for different circumstances or users [39]. In terms of *reproducibility*, a benchmark should provide run-to-run consistency in terms of results, as well as for a different user in an identical environment [39]. Run-to-run variability can be mitigated by performing workloads for long periods of time or executing them multiple times to effectively generate representative sample sizes [39]. *Reproducibility* over multiple environments might be achieved by providing adequate description of the test environment for both hardware and software, as well as used configuration options [39]. In terms of *Fairness*, a system should be able to compete in the benchmark without any artificial constraints [39]. Although it might be necessary to enforce restrictions in terms of technical requirements and hardware components, for example, a benchmark might require a certain number of CPU cores [39]. *Verifiability* ensures the

trustworthiness and correctness of the produced results [39]. Ideally, configuration details are documented by the benchmark itself in the result [39]. If documentation of configuration parameters is only done by the user, it tends to be less trustworthy due to the possibility of entering parameters incorrectly [39]. Finally, *usability* can, for example, be achieved by self-validating workloads or by providing tools that automatically extract the relevant information to generate the benchmark description [39].

4 Benchmarking setup

This chapter describes the general structure of the implemented benchmark cases, abstracting from the specifics regarding the alternative implementation options – native RDF-based and relational database management system. Section 4.1 focuses on the captured and used data and its structure as well as general design decisions regarding context granularity. In Section 4.2 used rules and corresponding derived facts are described. Finally, Section 4.3 covers requirements and their consideration in regard to materialization of previously derived facts.

4.1 Static and dynamic data

This section describes the difference between dynamic data and static data and how data is acquired, as well as the chosen contextualization approach for the used data. The data schema is separated into dynamic and static data. Dynamic data changes frequently over time, whereas static data is unchanging. In terms of dynamic data, the OpenSky-Network provides real-time data of current flights. For the benchmark, the data of these flights is captured periodically over Germany and saved per time instance. The captured data is used for all experiments to achieve benchmark reproducibility, as well as comparability between different configurations and architectures of the implemented systems. If a higher volume of data is required to test the performance of implemented systems, the captured data might be repeatedly duplicated. The OpenSky-Network also provides data regarding airports. As Schütz et al. [24] mention in their work about contextualized KGs in air traffic management, data in this domain is inherently context dependent, especially regarding the time dimension. However, for the purpose of the implemented benchmark, airport data is considered as unchanging and therefore static data. This enables a differentiation between static data and dynamic data while keeping the overall data schema complexity low. The captured dynamic flight data is time dependent, as it is periodically retrieved from the provided OpenSky-API. This time dependency is described via incrementally increasing time steps. Time steps represent an arbitrary slot of time that could be attributed to broader time intervals if the data schema allows for an aggregation over a hierarchy. Therefore, time steps represent one level of a context dimension of time. For the benchmark performed in this work, an aggregation to higher levels of a context dimension is not considered. Expansion of contextualization to multiple dimensions is also not investigated for benchmarking.

4.2 Rule implementations

In this section, the implemented rules are described in general. Two different rules were considered for benchmarking. The first is a distance calculation between airports and flights (*Rule 1*) and the second is a position prediction of an aircraft in the future based on current speed and trajectory values (*Rule 2*). In a preliminary study conducted by Steiner [41], queries and rules were already explored using various RDF-based technologies. This work builds on the results by Steiner [41].

4.2.1 Distance calculation – Rule 1

The distance between flights and airports is composed of the horizontal distance and the vertical distance. There are different ways to calculate distances in a geographical space, especially when taking calculation complexity into consideration. To maintain comparability to the results obtained in

the preceding study, similar distance calculation approaches are used. In his work Steiner, [41] focuses on two common ways to calculate the horizontal distance between geographical points. These are the Haversine distance and the equirectangular distance [41]. The equirectangular distance approximates the Haversine distance and yields less accurate results. However, due to the reduced complexity and an acceptable level of accuracy, the equirectangular distance is used, which is shown in Formula 1.

$$x = (lon2 - lon1) \times cos\left(\frac{lat1 + lat2}{2}\right)$$
$$y = lat2 - lat1$$
$$distance = \sqrt{x^2 + y^2} \times R$$

Formula 1: Calculation of equirectangular distance

Additionally, distance between airports and flights is not only concerning the horizontal but also the vertical distance between them. Vertical distance is independent of Earth's curvature and therefore just the difference of two altitude values. Overall distance is then calculated via the Euclidean distance using a combination of horizontal and vertical distance [41]. Steiner [41] captures the calculated distances in three different distance triples with triple predicates *horizontalDistance*, *verticalDistance* and *euclideanDistance*. Generally, this work follows a similar approach. However, in Section 8.4, different variants of this layout are still explored due to differences in performance.

4.2.2 Position prediction – Rule 2

The second rule describes a position prediction of an ongoing flight. The rule takes the current latitude, longitude, altitude, direction, and speed of an aircraft as input and derives its predicted position. The position of the aircraft is predicted by means of a time input of 10 seconds. The calculation of the position is based on the rules implemented by Steiner [41]. Steiner [41] implemented three different variants of this rule, each differing only in the point of time for which the prediction is performed. For the purpose of this work, only one of those variants is implemented for architectures and technologies in 5.3 and 8.4, as the performance difference between each variant is negligible.

Similar to the first rule, *Rule 2* also covers a horizontal and vertical aspect of flight positions. The derived facts are captured in three separate triples with the predicates *predictedLongitude*, *predictedLatitude* and *predictedAltitude* [41]. To maintain comparability to the preceding study, derived facts are also captured in three corresponding triples or columns when the rule is implemented for the native and relational systems. Calculations of predicted longitude, latitude, and altitude are shown in Formula 2, Formula 3 and Formula 4. Latitude and longitude predictions are again based on the equirectangular distance approach.

$$predLat = lat + toDegrees\left(\frac{velocity \times time \times cos(heading)}{earthRadius}\right)$$

Formula 2: Calculation of predicted latitude

$$predLon = lon + toDegrees\left(\frac{velocity \times time \times sin(heading)}{earthRadius \times cos(toRadians(lat))}\right)$$

Formula 3: Calculation of predicted longitude

$$predAlt = alt + (verticalRate \times time)$$

Formula 4: Calculation of predicted altitude

4.3 Handling dynamic data

Handling dynamic data concerns general requirements and considerations about the materialization and deletion of dynamic base data and derived facts. As future potential rules will use historic derived facts of rules, it is required to keep a minimum of 120 time steps available at any given time. Additionally, in this work it is assumed that the system requires 100% availability due to safety concerns in aviation. Therefore, it is not reasonable to shift maintenance jobs to a time when the system is out of order. It is expected that maintenance jobs will interfere with the performance of the system during business operations. This means a balance has to be achieved so that data maintenance work can be performed while still fulfilling the requirements of minimum data availability and performance. There are different approaches to handling dynamic data. However, each approach is strongly dependent on the applied architecture and technologies. Therefore, specific data handling approaches and their performance are more closely evaluated for the native and relational system.

4.4 Benchmarking pipeline

In this section, the general benchmarking pipeline is described. To obtain comparability between each implemented approach, the same benchmarking pipeline is used to conduct the performance experiments. The general benchmark pipeline is shown in Figure 5. Even though the general benchmark structure is similar for all architectures and used technologies, concrete step-by-step implementations differ and are described in detail for each alternative implementation. In general, benchmarking aims for relevance and reproducibility. The target audience for the conducted benchmarking are the project partners, who will base their decision about which technology to choose, partly on the results of the benchmark.

A specific system configuration is tested via repeatedly inserting prepared data of the OpenSkyNetwork. The data is at this point already contextualized in the time dimension via an attached timestep attribute. The data insertion then triggers the execution of reasoning rules according to the configuration under test. Then data handling tasks are performed, if required by a specific configuration. Rule execution and data handling time are tracked and logged. Insertion time might not be tracked for individual relational architectures as it can be easily excluded. The reason for exclusion is that insertion time strongly depends on the system and available representation of the base data in practice. It still plays a part in the final comparison of the native-RDF approach and the relational approach, but then assumptions over the available representation of data are made. The

complete insertion of all available graphs for a specific configuration is considered a *configuration test run*.

To increase reproducibility, previously described in Section 3.3, each *configuration test run* is executed 10 times. The sequence of performing the *configuration test runs* might differ between architectures or used technologies due to implementation restrictions. To maintain comparability between each *configuration test run*, the test environment is reset. Additionally, as the system is considered to be always running, it is essential to simulate a warm start for each test run. As some rules may depend on data from preceding time steps, to simulate a warm start after the test environment is reset, it is filled with data corresponding to 120 time steps before starting performance measurements. Even though, the workload changes between different *configuration test runs*, performance of the investigated technologies and architectures is still evaluated via consumed time. Therefore, the benchmark is considered a fixed-work benchmark.

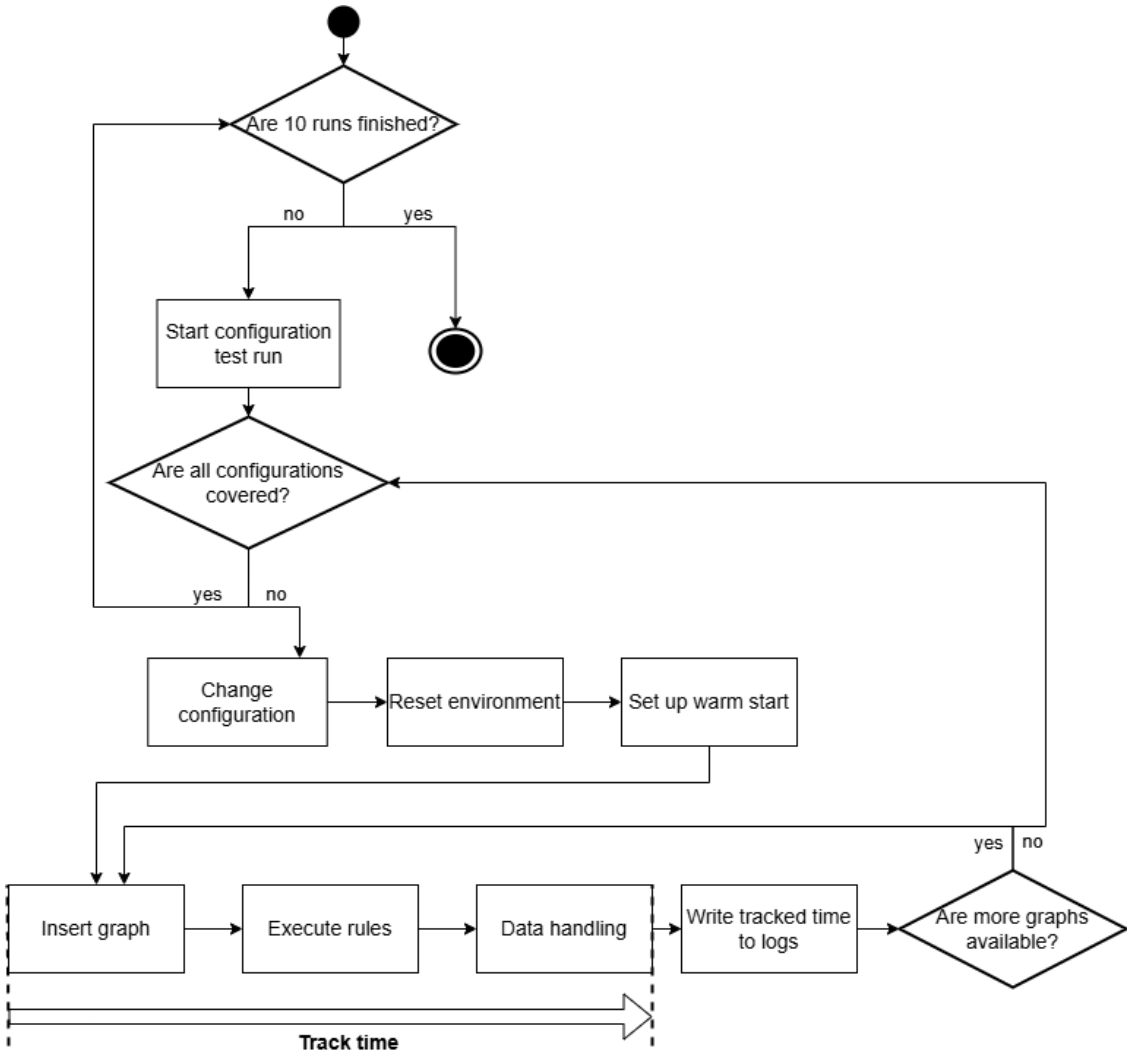


Figure 5: Benchmark structure

5 Native RDF-based KG management

This chapter describes the specific native RDF graph-based version. An RDF graph management system is employed as the underlying technology. First, the data schema is presented, followed by the datastore, a critical component within the RDF graph management system. Subsequently, querying and reasoning are discussed, with the two rules being defined in Datalog. The chapter on handling dynamic data outlines the architectures that enable continuous operation. The exact procedure for carrying out performance tests is detailed as well. Due to the restrictive nature of the license agreement, we are not allowed to publicly disclose the results of performance experiments.

5.1 Data schema

This chapter examines the structure of the data schema and explains the interaction between modules and Datalog rules. Furthermore, it discusses how the test data used for the experiments was represented in RDF. It is noted that an adapted Datalog syntax was employed.

5.1.1 Datalog rules

A Datalog rule is a logical rule used to derive new facts from existing ones. It is a central concept in logic programming and is frequently applied in databases and knowledge representation systems. Such a rule consists of two main components:

- **Rule Head**: The rule head describes the goal or conclusion of the rule. It represents the facts that are derived when the conditions in the rule body are satisfied.
- **Rule Body**: The rule body contains the conditions that must be met for the rule head to be derived. These conditions consist of one or more predicates, which may be interconnected.

The rule head and rule body are separated by the string ":-". The rule head is located on the left side of the separator, while the rule body is on the right side. Since this rule processes RDF facts, it must be specified where the facts to be derived originate. These facts can come from the default graph or from named graphs. If the facts originate from the default graph, no explicit name for the named graph needs to be provided. However, if the facts come from a named graph, the name of the named graph must be specified.

Listing 4 provides an example of a Datalog rule. As shown, facts that match the described pattern in the default graph are transferred to a named graph called "namedGraph1". It is also possible to specify a prefix, such as for "predicate" but this prefix must be explicitly registered in the RDF graph management system.

```
RULE{
  :predikat(subject,object) namedGraph1 :- :predikat(subject,object) .
  }
```

Listing 4: Example Datalog rule

5.1.2 Test data

In this section, the test data used for the performance tests is described in detail. The test data was generated using the OpenSky Network. For different points in time, referred to as timestamps, the corresponding data was retrieved, processed using a utility program, and stored in the Turtle syntax described in Chapter 3.2.1. Since the OpenSky Network occasionally provided null values, these null values were handled by omitting the generation of the affected triples. For each individual timestamp, a separate Turtle file was created to store the corresponding flight states. Airports served as static data and were also stored in a separate Turtle file. Listing 5 illustrates the representation of the properties of a flight state, while Listing 6 shows the representation of an airport.

```
@prefix : <http://aware-sample-data/flight#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
:FlightState-XRO1251-1721912825
      a               :FlightState ;
      :hasCallSign       "XRO1251" ;
      :hasIcao       "3ccc7f" ;
      :hasOriginCountry       "Germany" ;
      :hasFlightState       "1721912825" ;
      :hasSpecialInfo       "" ;
      :hasAltitude       "5181.6"^^xsd:float ;
      :hasGeoAltitude     "5417.82"^^xsd:float ;
      :hasHeading        "71.48"^^xsd:float ;
      :hasLatitude       "47.5086"^^xsd:float ;
      :hasLongitude       "9.3028"^^xsd:float ;
      :hasPositionSource  0 ;
      :hasSquawk         "1000" ;
      :hasVelocity       "208.88"^^xsd:float ;
      :hasVerticalRate    "0.33"^^xsd:float ;
      :isOnGround        false ;
      :isSpi           false .
```

Listing 5: Representation of a flight state in turtle syntax.

```
@prefix :    <http://aware-project/adsb#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix pointer: <http://aware-pointer/to#> .
:Airport-EDCL  a      :Airport ;
      :hasContinent  "EU" ;
      :hasCountry    "Germany"@en ;
      :hasElevation  "28.956001"^^xsd:float ;
      :hasGpsCode    "EDCL" ;
      :hasLatitude   "52.709442"^^xsd:float ;
      :hasLongitude  "12.073333"^^xsd:float ;
      :hasName       "Klietz/Scharlibbe Airfield" ;
      :hasRegion     "Saxony-Anhalt"@en ;
      :hasType       "small_airport" .
```

Listing 6: Airport representation in turtle syntax

5.1.3 Modules

To contextualize the facts, abstract modules are introduced based on the AISA project. These modules aim to simplify the management of the knowledge graph (KG) for both the user and the producer in relation to the Datalog rules mentioned in the section 5.3

These modules can be categorized into internal and external modules, which can further be divided into static and dynamic modules. A static module refers to a module where the underlying data remains unchanged. Dynamic modules, on the other hand, are used for data that is recurring and has a temporal context. Each internal dynamic module can include a set of specific Datalog rules, as described in Chapter 5.3. It is noted that in the architectures described in the sections 5.4.1 and 5.4.2, only modules are implemented, and this distinction has not been applied, as it does not play a critical role in the performance tests. However, this module management structure can be implemented with minimal effort. If one chooses to forgo this modular structure, it is also possible to use a single module that encompasses all rules and data sources.

5.1.4 Datastores

The employed RDF graph management system can provide multiple datastores. Within a datastore, it is possible to manage multiple tuple tables, which can store either N-Quads or simple RDF triples. Queries on these RDF triples can be executed using SPARQL queries. Additionally, it is possible to retrieve facts from a datastore through alternative methods. The RDF graph management system also allows the creation of transactions, which are categorized into write and read transactions. Users can create such transactions, perform queries or insert data within the transaction, and subsequently close them. A single datastore supports single-writer (exclusive write access) and multiple-reader (concurrent read access) modes for multi-user control. It should be noted that maintenance operations can be performed, which result in an exclusive lock state. Due to this limitation, the experiments conducted examined the import and reasoning capabilities of the respective architectures in conjunction with the underlying RDF graph management system. This was necessary, because data from multiple modules must be written to the datastore, and new facts must be derived. However, this process cannot occur in parallel due to the single-writer restriction. The maintenance operations, which require an exclusive lock, can significantly impair the system. According to current knowledge, it is not possible to avoid or bypass these operations during continuous operation.

5.2 Querying

As described in Chapter 5.1.4, facts can be queried from the datastore using SPARQL. Through the SPARQL HTTP endpoint, arbitrary RDF triples or N-Quads can be retrieved or inserted. These queries can be processed in parallel. Each query is executed within a transaction. If the user submits a SPARQL query via the HTTP endpoint without specifying a transaction, the system automatically creates a transaction, which is closed immediately after the query execution. Due to this behaviour, all architectures omit active transaction management, as it would not provide any apparent benefit.

5.3 Rule formulation

The specific rules implemented to derive new knowledge from the inserted data can be formulated statically or dynamically. In a dynamic formulation, the name of the utilized named graph is logically derived at runtime and thus determined dynamically. In the static formulation, the named graph is already fixed within the rule definition and therefore does not need to be derived during rule execution within the RDF graph management system.

To perform calculations in Datalog, the built-in functions of the RDF graph management system in use are utilized. Compared to the relational approach this system does not allow the implementation of external functions—only the built-in functions are available. For this reason, it was not possible to further simplify the Datalog rules formulated in the following sections. The exact execution of the mentioned calculations is not described in detail; Section 4.2 provides an explanation of the logic of the formulated rules.

5.3.1 Distance calculation (Rule 1)

This rule focuses on distance calculations. It computes the distances between an airport and all airplanes currently in the air. Three different types of distances are calculated: the horizontal distance, the vertical distance, and the total distance. It becomes evident with this rule that the number of derived facts depends on the number of airports and airplanes, which has a significant impact on the rule's execution time. This rule was also selected because the number of derived facts can be scaled very easily and significantly by increasing the number of airports.

Rule 1 – statically formulated

This paragraph describes the formulation of the static version. Listing 7 presents the rule in its static version.

In this approach, as previously described, the names of the named graphs from which the facts for the derivation are to be retrieved are determined prior to the actual rule execution. As can be seen, a specific syntax was defined for the static version regarding the names of the named graphs from which the data is to be used. This is important to enable targeted resolution of the names later in the corresponding proxy of the respective architecture. The name resolution, as well as the chosen structure of the names, is described in greater detail in Chapter 5.4.1. As a result, it is already determined at the time the rule is inserted into the datastore which facts will be used for the derivation. The static data used in the rule—in this case, airports—are directly assigned a corresponding prefix in their names. This is possible because airports, as previously mentioned, are considered static data that exist in the datastore once and do not change dynamically at runtime. This is evident in Listing 7 with the static data highlighted in **green** and the dynamic and derived facts highlighted in **red**. It is important to emphasize at this point that the name of a named graph is not logically derived within the rule but rather transformed. The prefixes used must already exist at the time the rule is introduced.

```
RULE{
:AirportDistance(d) <http://www.aware-project.eu/current-0/rule2>,
 :flight(d, f) <http://www.aware-project.eu/current-0/rule2>,
 :airport(d, airport) <http://www.aware-project.eu/current-0/rule2>,
 :horizontalDistance(d, horizontalDistance) <http://www.aware-project.eu/current-0/rule2>,
 :verticalDistance(d, verticalDistance) <http://www.aware-project.eu/current-0/rule2>,
 :euclideanDistance(d, euclideanDistance) <http://www.aware-project.eu/current-0/rule2> :-

 :FlightState(sf) <http://www.aware-project.eu/current-0/airData>,
 :flight(sf, f) <http://www.aware-project.eu/current-0/airData>,
 :hasLatitude(sf, lat1) <http://www.aware-project.eu/current-0/airData>,
 :hasLongitude(sf, lon1) <http://www.aware-project.eu/current-0/airData>,
 :hasGeoAltitude(sf, alt1) <http://www.aware-project.eu/current-0/airData>,
 :isOnGround(sf, false) <http://www.aware-project.eu/current-0/airData>,

 :Flight(f) <http://www.aware-project.eu/current-0/airData>,
 :hasCallSign(f, callsign) <http://www.aware-project.eu/current-0/airData>,

 :Airport(airport) pointer:airports,
 :hasGpsCode(airport, gpsCode) pointer:airports,
 :hasLatitude(airport, lat2) pointer:airports,
 :hasLongitude(airport, lon2) pointer:airports,
 :hasElevation(airport, alt2) pointer:airports,

 BIND(lat1 * PI() / 180 AS lat1_rad),
 BIND(lat2 * PI() / 180 AS lat2_rad),
 BIND(lon1 * PI() / 180 AS lon1_rad),
 BIND(lon2 * PI() / 180 AS lon2_rad),

 BIND((lon2_rad - lon1_rad) * COS((lat1_rad + lat2_rad) / 2) AS x),
 BIND(lat2_rad - lat1_rad AS y),
 BIND((xsd:float(SQRT(x * x + y * y) * 6371)) AS horizontalDistance),
 BIND(xsd:float(ABS(alt2 - alt1)) AS verticalDistance),
 BIND(xsd:float(SQRT(POW(horizontalDistance, 2) + POW((verticalDistance / 1000), 2))) AS euclidean
Distance),
 BIND(IRI(CONCAT(STR(:), "AirportDistance-", STR(callsign), "-", STR(gpsCode))) AS d). }
```

Listing 7: Rule 1 statically formulated

Rule 1 – dynamically formulated

This paragraph describes the formulation of the dynamic version. The rule presented in Listing 8 generates the same facts as the rule shown in Listing 7. In contrast to the static variant, the name of the named graph is logically determined within the rule in this case. This is shown in lines highlighted in **green** in Listing 8. The corresponding insertion of the facts from which these names for the named graphs are derived is described in Chapter 5.4.2. Compared to the static variant, the name of the named graph, where the respective derived facts are stored, is only determined during the execution of the rule. In green highlighted lines placeholder "kkk" appears twice. It is important to note that

these are merely placeholders to facilitate the execution of performance tests. During the actual implementation, the placeholder "kkk" values were replaced with natural numbers.

The ability to access facts from the past is enabled by the expression "BIND(?index-kkk AS ?index_ta)" highlighted in green. In this rule formulation, there is no need to reference the current timestamp, as the rule is applied to *all* facts in the datastore. Furthermore, apart from the operations related to determining the names of the named graphs, the operations in this formulation are identical to those in the static rule formulation.

```
RULE{
:AirportDistance(d) module_name,
 :flight(d, callsign) module_name,
 :airport(d, airport) module_name,
 :horizontalDistance(d, horizontalDistance) module_name,
 :verticalDistance(d, verticalDistance) module_name,
 :euclideanDistance(d, euclideanDistance) module_name :-

 timestamp:hasModule0(ta, name),
 timestamp:index(ta, index_ta),
 timestamp:hasModulekkk(s, module_name),
 timestamp:index(s, index),
 :FlightState(fs) name,
 :hasLatitude(fs, lat1) name,
 :hasLongitude(fs, lon1) name,
 :hasGeoAltitude(fs, alt1) name,
 :isOnGround(fs, false) name,
 :hasCallSign(fs, callsign) name,
 :Airport(airport) pointer:airports,
 :hasGpsCode(airport, gpsCode) pointer:airports,
 :hasLatitude(airport, lat2) pointer:airports,
 :hasLongitude(airport, lon2) pointer:airports,
 :hasElevation(airport, alt2) pointer:airports,
 BIND(index-kkk AS index_ta),
 BIND(lat1 * PI() / 180 AS lat1_rad),
 BIND(lat2 * PI() / 180 AS lat2_rad),
 BIND(lon1 * PI() / 180 AS lon1_rad),
 BIND(lon2 * PI() / 180 AS lon2_rad),
 BIND((lon2_rad - lon1_rad) * COS((lat1_rad + lat2_rad) / 2) AS x),
 BIND(lat2_rad - lat1_rad AS y),
 BIND((xsd:float(SQRT(x * x + y * y) * 6371)) AS horizontalDistance),
 BIND(xsd:float(ABS(alt2 - alt1)) AS verticalDistance),
 BIND(xsd:float(SQRT(POW(horizontalDistance, 2) + POW((verticalDistance / 1000), 2))) AS euclidean
Distance),
 BIND(IRI(CONCAT(STR(:), "AirportDistance-", STR(callsign), "-", STR(gpsCode))) AS d).}
```

Listing 8: Rule 1 dynamically formulated

5.3.2 Position prediction (Rule 2)

In this rule, the prediction of a flight is calculated. For each flight, a new anticipated flight status is derived within this rule. Like Rule 1, these are stored in a separate named graph. The predicted state consists of the flight status, latitude, longitude, altitude, and time. Only the data of the flight or flight status is required for deriving these facts. The number of derived facts is linearly dependent on the number of flights. A distinction can again be made between a static and a dynamic rule description.

Rule 2 – statically formulated

In Listing 9, Rule 2 is formulated in the static variant. A detailed description is not provided here, as this procedure has already been explained in section 5.3.1.

```
RULE{
:PredictedState(predictedState) <http://www.aware-project.eu/current-0/rule5>,
 :hasFlightState(predictedState, fs) <http://www.aware-project.eu/current-0/rule5>,
 :hasLatitude(predictedState, pred_lat) <http://www.aware-project.eu/current-0/rule5>,
 :hasLongitude(predictedState, pred_lon) <http://www.aware-project.eu/current-0/rule5>,
 :hasAltitude(predictedState, pred_alt) <http://www.aware-project.eu/current-0/rule5>,
 :hasTime(predictedState, time) <http://www.aware-project.eu/current-0/rule5> :-

 :Flight(f) <http://www.aware-project.eu/current-0/airData>,
 :hasCallSign(f, callsign) <http://www.aware-project.eu/current-0/airData>,
 :FlightState(fs) <http://www.aware-project.eu/current-0/airData>,
 :flight(fs, f) <http://www.aware-project.eu/current-0/airData>,
 :hasLatitude(fs, lat) <http://www.aware-project.eu/current-0/airData>,
 :hasLongitude(fs, lon) <http://www.aware-project.eu/current-0/airData>,
 :hasGeoAltitude(fs, alt) <http://www.aware-project.eu/current-0/airData>,
 :hasHeading(fs, heading) <http://www.aware-project.eu/current-0/airData>,
 :hasVelocity(fs, velocity) <http://www.aware-project.eu/current-0/airData>,
 :hasVerticalRate(fs, verticalRate) <http://www.aware-project.eu/current-0/airData>,
 :isOnGround(fs, false) <http://www.aware-project.eu/current-0/airData>,
 BIND(10 AS time),
 BIND(IRI(CONCAT(STR(:), "PredictedState-", STR(callsign), "-", STR(time))) AS predictedState),
 BIND(velocity * time AS distance),
 BIND(heading * PI() / 180 AS heading_rad),
 BIND(distance * COS(heading_rad) / 6371000 AS delta_lat),
 BIND(delta_lat * 180 / PI() AS delta_lat_deg),
 BIND(xsd:float(lat + delta_lat_deg) AS pred_lat),
 BIND(lat * PI() / 180 AS lat_rad),
 BIND(distance * SIN(heading_rad) / (6371000 * COS(lat_rad)) AS delta_lon),
 BIND(delta_lon * 180 / PI() AS delta_lon_deg),
 BIND(xsd:float(lon + delta_lon_deg) AS pred_lon),
 BIND(verticalRate * time AS delta_alt),
 BIND(xsd:float(alt + delta_alt) AS pred_alt).
   }
```

Listing 9: Rule 2 statically formulated

Rule 2 – dynamically formulated

In Listing 10, Rule 2 is formulated in the dynamic variant. A detailed description is not provided here, as this procedure has already been explained in section 5.3.1.

```
RULE{
   :PredictedState(predictedState) module_name,
 :hasFlightState(predictedState, fs) module_name,
 :hasLatitude(predictedState, pred_lat) module_name,
 :hasLongitude(predictedState, pred_lon) module_name,
 :hasAltitude(predictedState, pred_alt) module_name,
 :hasTime(predictedState, time) module_name :-

 timestamp:hasModule0(ta, name),
 timestamp:index(ta, index_ta),
timestamp:hasModulekkk(s, module_name),  timestamp:index(s, index),

 :hasCallSign(fs, callsign) name, :FlightState(fs) name, :hasLatitude(fs, lat) name,
 :hasLongitude(fs, lon) name, :hasGeoAltitude(fs, alt) name,
 :hasHeading(fs, heading) name,:hasVelocity(fs, velocity) name,
 :hasVerticalRate(fs, verticalRate) name, :isOnGround(fs, false) name,
 BIND(index - xxx AS index_ta),
 BIND(10 AS time),
 BIND(IRI(CONCAT(STR(:), "PredictedState-", STR(callsign), "-", STR(time))) AS predictedState),
 BIND(velocity * time AS distance),
 BIND(heading * PI() / 180 AS heading_rad),
 BIND(distance * COS(heading_rad) / 6371000 AS delta_lat),
 BIND(delta_lat * 180 / PI() AS delta_lat_deg),
 BIND(xsd:float(lat + delta_lat_deg) AS pred_lat), BIND(lat * PI() / 180 AS lat_rad),
 BIND(distance * SIN(heading_rad) / (6371000 * COS(lat_rad)) AS delta_lon),
 BIND(delta_lon * 180 / PI() AS delta_lon_deg),
 BIND(xsd:float(lon + delta_lon_deg) AS pred_lon),
 BIND(verticalRate * time AS delta_alt),
 BIND(xsd:float(alt + delta_alt) AS pred_alt).
}
```

Listing 10: Rule 2 dynamically formulated

5.4 Handling dynamic data

In the following, we describe the handling of dynamic data. Two architectures are presented that can efficiently import dynamic data. The proposed architectures are designed to meet real-time criteria and enable continuous operation under a certain load. To achieve this, it is necessary to delete outdated data, as it is not feasible to store all historical facts in memory during continuous operation.

The architectures were subsequently tested for their performance, and these performance tests are described in Section 5.5.7.

These architectures include a proxy responsible for managing the rules and modules. This ensures that switching between architectures can be done without requiring significant adjustments. In the following chapters, these two architectures are described in greater detail, and their functionality is explained. The focus is placed on the theoretical approach rather than the specific implementation, which was carried out in Java to enable the experiments.

5.4.1 Architecture 1

This section aims to describe the first architecture. First, this section outlines the fundamental functionality, followed by an explanation of the management of Datalog rules. In addition, the process of inserting the corresponding dynamic data is described, along with the locations within the proxy where measurement points are situated.

Fundamental functionality

This architecture requires the procedure illustrated in Figure 6 to function correctly. Within this process, there is an interval during which the data of new timestamps must be continuously inserted. The comments Figure 6 describe the operations being performed. The mentioned data labels, such as "Data1," abstractly represent all data associated with a specific timestamp.

It is crucial to regularly delete outdated data to ensure that new facts can be inserted without conflicts. Additionally, it is important that this data is structured and stored in the corresponding named graphs in such a way that rules can later interact with them.

Intervall

Comment	0	1	2
Startpoint	null	null	null
Insert TS1	Data1	null	null
Insert TS2	Data1	Data2	null
Insert TS3	Data1	Data2	Data3
Insert TS4	Data4	Data2	Data3
Insert TS5	Data4	Data5	Data3

Figure 6: Schema for Timestamp Insertion

Co-funded by the European Union

Management of Datalog rules

This chapter explains how the proxy handles the corresponding Datalog rules. This architecture processes statically formulated rules. Chapter 5.1.3 describes that the data from each module is stored in its own named graph. This is necessary to properly configure the proxy. Proper configuration is important because a separate rule set is defined for each timestamp within the storage interval.

The following example is provided to better illustrate this process.

In this example, there are three modules. Two of these modules are external data sources that supply data to the datastore but do not have any rules. These modules are named "M1" and "M2." The third module, named "M3," contains a rule, which is shown in Listing 11. In this rule, "s1" from the module "M1" at a timestamp in the past and "o2" from the module "M2" at two timestamps in the past are to be linked into a new fact.

The following sections explain how this rule is processed by the proxy and the role that the naming of the named graphs plays in this process.

```
RULE{
  p1(s1,o2) <k/current-0/M3> :- p1(s1,o1) <k/current-1/M1>, p2(s2,o2) <k/current-2/M2>
  .
  }
```

Listing 11: Sample rule for the example

The name of a named graph containing dynamic data follows the structure described in Listing 12. It should be noted that the outer angle brackets, which always enclose the name, are omitted here to focus solely on the name itself:

```
<unique>/<relative>/<module>
```

Listing 12: Structure of the Named Graph Name

This structure can be broken down as follows:

- **<unique>**: Serves as a unique identifier.
- **<relative>**: Indicates the relative position to the current timestamp. This part of the name is particularly important for the transformation of the rule. The <relative> string is composed of a fixed prefix, "current-", followed by a natural number. This number specifies how many timestamps in the past one needs to go back. It should be noted that this value must be smaller than the storage interval.
- **<module>**: Specifies the name of the module.

During the transformation of the rule, the relative position to the current timestamp is converted into an absolute position. This conversion process is of particular importance. Figure 7 illustrates the naming table. It shows how the named graphs for the individual modules are labeled in the respective stored timestamps.

Co-funded by the European Union

Timestamp	M1	M2	M3
0	k/0/M1	k/0/M2	k/0/M3
1	k/1/M1	k/1/M2	k/1/M3
2	k/2/M1	k/2/M2	k/2/M3
3	k/3/M1	k/3/M2	k/3/M3

Figure 7: Name table

If we now transform the rules, we obtain the rule sets depicted in Listing 13. These specific rules can then be inserted into the datastore. Each timestamp thus has its own rule set.

The total number of actual rules stored by the proxy is determined by multiplying the number of modules by the number of timestamps to be stored.

```
#Timestamp 0
RULE{
   p1(s1,o2) <k/0/M3> :- p1(s1,o1) <k/3/M1>, p2(s2,o2) <k/2/M2>
   .
   }
#Timestamp 1
RULE{
   p1(s1,o2) <k/1/M3> :- p1(s1,o1) <k/0/M1>, p2(s2,o2) <k/3/M2>
   .
   }
#Timestamp 2
RULE{
   p1(s1,o2) <k/2/M3> :- p1(s1,o1) <k/1/M1>, p2(s2,o2) <k/0/M2>
   .
   }
#Timestamp 3
RULE{
   p1(s1,o2) <k/3/M3> :- p1(s1,o1) <k/2/M1>, p2(s2,o2) <k/1/M2>
   .
   }
```

Listing 13: Set of created Datalog rules for each Timestamp

Co-funded by the European Union

Insertion of dynamic data

This section describes how the dynamic data intended for storage in the datastore is inserted into the datastore of the RDF graph management system via the proxy. This insertion triggers a series of auxiliary processes to handle the corresponding tasks. In this version, it must first be determined into which datastores the data should be inserted. This determination is illustrated in Figure 8.

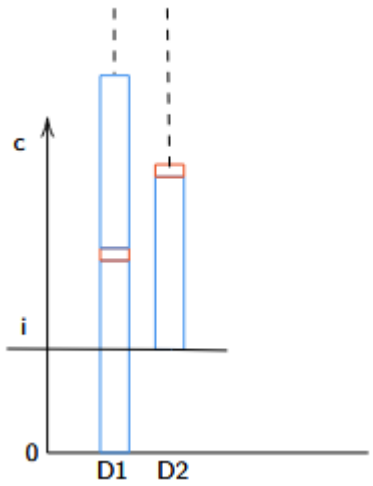


Figure 8: A visual representation of datastore handling

The functionality of using two datastores is illustrated in Figure 8. D1 and D2 represent the two datastores, while *c denotes* the continuous timestamps. A blue field indicates the period during which a datastore is actively used, while an orange field represents a switching period. The variable *i* refers to the storage interval, with *c* initially set to zero. A switch occurs whenever the condition defined in the formula is met. There is a current (active) datastore and a non-current (inactive) datastore; initially, the current datastore is D1. Starting from interval *i*, data is written not only to D1 but also to D2. Whenever the blue rectangles of D1 and D2 overlap, the same data is processed in both datastores in parallel. This partial parallel operation is crucial to ensure that the historical data remains available when the switch occurs. The reason for this parallel operation, rather than using a shadow copying approach, is performance. If derived facts were copied, the rule execution would need to be completed first, followed by querying the data and then inserting it into the other datastore. Querying and inserting data takes time and is therefore slower than parallel operation, where facts are derived simultaneously in both datastores without the need for querying or inserting. Before dynamic data can be inserted into a datastore, it must first be populated with static data. This occurs during the switching period. Prior to inserting dynamic data, the datastore is deleted and recreated to quickly remove outdated dynamic data.

Once it has been determined into which datastores the data will be inserted, the following operations must be performed before the actual insertion of dynamic data for a timestamp can take place:

1. **Committing the Derived Facts and Deleting the Datalog Rules in the Datastore**: The described RDF graph management system distinguishes between facts that have been inserted and those that have been derived. As previously mentioned, a slot in the interval is overwritten only once, making it necessary to perform this operation.

2. **Deleting Outdated Data in the Corresponding Interval Slot**: This operation is essential to ensure that no outdated data is stored in a named graph.
3. **Inserting the Rule Set**: Each timestamp in the interval has its own rule set, which must be inserted.

Once these operations have been successfully completed, the corresponding dynamic data for a timestamp can be inserted.

5.4.2 Architecture 2

This chapter provides a detailed description of Architecture 2. Unlike Architecture 1, this architecture utilizes dynamic rules instead of static rules.

Management of Datalog rules

This chapter provides a detailed description of the management of Datalog rules. Dynamic rules offer the advantage that a rule only needs to be formulated once and can then be reused. This means that the rule set exists only once in the proxy, eliminating the need for dynamic transformation of rules within the proxy, as required in Architectures 1 and 2. These rules are inserted into the datastore at the beginning, when a new datastore is created.

Insertion of dynamic data

The insertion process can be divided into several auxiliary operations in addition to the actual insertion of dynamic data into the datastore. These auxiliary operations are essential for the continuous operation of the application and must be performed prior to the actual data insertion. They can be categorized as follows:

- Insertion of new metadata
- Deletion of old metadata for a specific timestamp.

In this context, metadata contains information required for deriving new facts. As already discussed in the description of dynamic rules, it is necessary to specify from which named graph the corresponding facts should be retrieved. In the specific implementation of the proxy, the SPARQL query shown in Listing 14 was used.

```
PREFIX timestamp: <http://aware-timestamp/>
PREFIX index: <http://index#>
PREFIX hasModule: <http://aware-hasModule/>
PREFIX module0: <http://www.aware-project.eu/airData/>
PREFIX module1: <http://aware-module1#>
PREFIX module2: <http://aware-module2#>
PREFIX module3: <http://aware-module3#>
PREFIX module4: <http://aware-module4#>
PREFIX module5: <http://aware-module5#>
PREFIX module6: <http://aware-module6#>
PREFIX module7: <http://aware-module7#>
PREFIX module8: <http://aware-module8#>
```

```
PREFIX module9: <http://aware-module9#>
PREFIX module10: <http://aware-module10#>
PREFIX module11: <http://aware-module11#>
PREFIX module12: <http://aware-module12#>
PREFIX module13: <http://aware-module13#>
PREFIX module14: <http://aware-module14#>
PREFIX module15: <http://aware-module15#>
PREFIX module16: <http://aware-module16#>
PREFIX module17: <http://aware-module17#>
PREFIX module18: <http://aware-module18#>
PREFIX module19: <http://aware-module19#>
PREFIX module20: <http://aware-module20#>


DELETE WHERE {
   timestamp:""" + (indexDel) + " timestamp:isCurrent   true ." +"""
}
INSERT DATA {
   timestamp:""" + index + " timestamp:hasModule0   module0:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule1   module1:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule2   module2:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule3   module3:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule4   module4:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule5   module5:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule6   module6:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule7   module7:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule8   module8:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule9   module9:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule10   module10:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule11   module11:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule12   module12:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule13   module13:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule14   module14:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule15   module15:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule16   module16:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule17   module17:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule18   module18:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule19   module19:" + index+" ." +"""
   timestamp:""" + index + " timestamp:hasModule20   module20:" + index+" ." +"""
   timestamp:""" + index + " timestamp:index  " + index+" ." +"""
   timestamp:""" + index + " timestamp:isCurrent   true ." +"""
}
```

Listing 14: Timestamp insert query

The “index” used in the query is incrementally increased to ensure that the names of the named graphs remain unique. In this implementation, a timestamp possesses the following properties:

- **Module**: This stores the names of the named graphs. In the illustration, these correspond to the modules of the system.
- **Index**: This index ensures that, when retrieving data from the timestamp externally, the index is available to simplify the formulation of the retrieval query. Additionally, it facilitates navigation to previous timestamps.
- **Current**: This indicates which timestamp is the most recent one.

Simultaneously, the SPARQL delete operation removes the “Current” entry from the preceding timestamp. This is necessary to ensure that there is always only one “Current” timestamp at any given time.

It should be noted that there are various implementation approaches, and in this case, a "hardcoded" variant was chosen for efficiency reasons. Additionally, 20 modules were added to avoid any limitations during subsequent performance tests related to rule duplication, as discussed in Section 5.5.7. It is worth mentioning that this architecture allows for significant expansion of metadata, as these additional facts have no impact on rule execution unless they are directly involved in it. Furthermore, the prefixes, which are also shown in Listing 14, can be modified as needed. However, it is crucial that these metadata align with the corresponding rule implementation. Without proper coordination, the metadata become useless, and deriving meaningful facts becomes challenging.

After these operations, the actual dynamic data can be inserted. These dynamic data are forwarded by the proxy to the KG management system. During this insertion process, the derivation of new facts occurs based on the corresponding rules already present in the datastore.

Datastore switching

As previously mentioned, this architecture again utilizes two datastores, which operate in almost the same sequence as previously described for Architecture 1, but with a slight difference. Unlike the process outlined previously, in this case, not only the static data but also the rules are inserted. This ensures that the rules are already available when the dynamic data are inserted.

5.5 Setup of performance experiments

The architectures described in Chapter 5.4 were evaluated in terms of their performance. To this end, a series of experiments was conducted on the respective architectures to enable a subsequent comparison of their performance. This section outlines the setup, including where and how the experiments were conducted, as well as a detailed description of the experiments themselves.

5.5.1 Virtual machine

For testing purposes, a virtual machine (VM) was set up. This VM uses Debian version 12 (Bookworm) as its operating system. The VM is allocated 100 gigabytes (GB) of memory, a value chosen to accommodate the in-memory operation of the specified KG management system. The hard disk size was set to 128 GB, as it plays a minimal role in performance measurements due to the in-memory nature of the KG management system. The number of cores (vCPUs) was set to 24, as the KG management system operates partially in parallel, and the experiments varied the number of cores to assess the impact of this parallelization. A maximum of 16 cores was assigned to the KG management

system. However, 24 cores were chosen for the VM to avoid resource conflicts between the KG management system and the programs described in Chapter 5.5.2.

The VM was operated on the following host system. The system is equipped with two AMD EPYC 7313 processors, each featuring 16 cores and 32 threads with a base frequency of 3.0 GHz, providing high parallel computing performance. The memory of the host system amounts to 2 terabytes (TB).

5.5.2 Used software during the performance tests

To conduct the experiments, appropriate testing software was required. This includes proxies, test runners, and various shell scripts. It should be noted that each architecture has its own proxy, and there are different implementations of the test runner depending on the rule formulation (static/dynamic). The various proxies were written in Java and executed in the corresponding Java environment (version 17) on a virtual machine (VM). The test runners, on the other hand, were operated on a desktop computer running Windows 11 also on Java 17. A detailed description of this system is omitted, as performance data for insert operations was also measured a second time on the proxy side. The test runners and their implementation are described in greater detail in Chapter 5.5.4.

As described in Chapter 5.5.7, a large number of experiments were conducted. To ensure the testing procedure outlined in Chapter 5.5.6 was followed for each experiment, a shell script was created to automate this process. This shell script will not be discussed in detail, as it merely facilitated the operational execution of the experiments.

5.5.3 General description of the measurement procedures

This section provides a general overview of how performance measurements were taken. Log4J was used for measurement. To generate meaningful and easily processable log data, a custom Logback configuration file was created specifically for the test runner. The test runner is composed of various Java classes. In each class where the execution time of specific code sections needed to be measured, a logger was instantiated and used for logging. Log entries were created at both the start and end of each measurement section, allowing the time interval between these entries to be calculated and, consequently, the execution time of the corresponding code section to be determined. The time required for the logging process itself is considered negligible, as its overhead is minimal compared to the duration of the measured operations. For control logs on the proxy side during insert operations, the standard logging framework of Spring Boot was employed, configured with a custom Logback setup. It is important to note that the objective was not to achieve millisecond-level precision in performance measurements but rather to maintain a high-level perspective on performance trends.

5.5.4 General description of test runner

This section provides a more detailed explanation of the test runners. A test runner is a Java program designed to automate the execution of experiments. Each rule architecture has its own specific implementation of the test runner. The primary responsibilities of a test runner include the following:

1. Loading the experiment parameters
2. Configuring the test parameters on the proxy (e.g., module creation and provisioning of reasoning rules)

3. Importing static data into the proxy
4. Importing and measuring dynamic data (timestamps) in the proxy or RDF graph system
5. Executing test queries to validate the import of dynamic data

The description of the proxy focuses solely on the measurement points. Methods that are not directly related to this process are intentionally omitted from the discussion.

5.5.5 Recording measurements in the test runner

As previously described in Chapter 5.5.3, the measurements focused on recording the execution time of individual code sections. Before each insert operation, test queries are executed, as shown in Listing 15.

These three queries were executed prior to the insertion of a timestamp, with the exception that "Query 3" was only executed in Architecture 3. The first query (Query 1) retrieves the total number of triples stored in the RDF graph. This serves two purposes: first, it allows the performance of a query involving an aggregate function to be evaluated, and second, it enables tracking changes in the number of triples during the test run. The second query (Query 2) retrieves the number of named graphs. This is used to verify whether the insertion and reasoning processes are functioning correctly throughout the experiment. Additionally, this query provides an opportunity to assess query performance. The third query retrieves a larger set of triples from the datastore. Its purpose is to evaluate the time required for such a retrieval operation, providing insights into the performance of large-scale data access. However, these measurement results were not extensively visualized, as they were only used to verify the functionality of the architecture and to quickly ensure that data could be queried from the datastore efficiently. Every query time was below one second!

```
#Query 1
SELECT (COUNT(*) AS ?count)
WHERE {
   GRAPH ?graph {
      ?s ?p ?o
   }
}
#Query 2
SELECT (COUNT(DISTINCT ?graph) AS ?graphCount)
WHERE {
   GRAPH ?graph {
      ?s ?p ?o
   }
}
#Query 3
PREFIX timestamp: <http://aware-timestamp/>

SELECT ?s ?p ?o
WHERE {
   ?x  timestamp:hasModule0 ?g .
   ?x  timestamp:isCurrent ?k .
   GRAPH ?g {
      ?s ?p ?o .
   }
}
```

Listing 15: Queries before insert operation

Furthermore, the insertion of facts into the datastore, along with the associated reasoning process, is measured. This insertion process is accompanied by the derivation of new facts, inferred based on the corresponding rules. Consequently, the derivation of new facts is also measured as part of this process. This measurement section is illustrated in Listing 16, which represents a segment of a much larger Java method. The full method is not described here, as it is not relevant to the context. It is important to note that the feedback from the RDF system is not analyzed, as its content does not contribute any additional value to the measurement process. Logtext markers (“insert;begin”, “insert;end”) were used to facilitate the later analysis of the log data in an efficient and straightforward manner.

```
logger.info("insert;begin;"+i+";"+a+";"+c);
try (CloseableHttpResponse response = httpClient.execute(addFactsRequest)){
                logger.info("insert;end;"+i+";"+a+";"+c);
```

Listing 16: Insert measurement section

5.5.6 Test procedure

To ensure that the experiments were conducted independently of one another, the procedure for each individual run was structured as follows:

1. Start the KG management system with the corresponding configuration.
2. Start the proxy.
3. Start the test runner with the specified experiment parameters.
4. After the experiment concludes: Copy the log data from the proxy, and test runner into a separate directory.
5. Shut down the KG system and proxy (Testrunner finish alone).

Each experiment was conducted ten times to mitigate the impact of external factors, such as operating system processes, on the results.

5.5.7 Specific experiments

A variety of experiments were conducted. Each experiment consisted of a set of different parameters. These parameters were carefully adjusted to align with the performance test setting and the rules outlined in Section 5.3. The following problem dimensions were defined:

1. **Airports**: In this dimension, the number of airports is varied. As described in Chapter 5.3.1, the number of airports directly influences the number of facts that need to be derived, as it is proportional to the number of airplanes.
2. **Airplanes**: The number of airplanes varies from one timestep to the next. This is because the test data, as described in Chapter 5.1.2, was taken directly from real-world data without modification to remain as close to reality as possible. The number of airplanes fluctuates slightly around the value of 655.
3. **Rule 1**: This parameter specifies whether Rule 1 should be applied in an experiment.
4. **Rule 2**: Similarly, this parameter specifies whether Rule 2 should be applied in an experiment.
5. **Rule Duplication**: This parameter is used to vary the number of rules. It specifies how many times a rule is duplicated. Duplication in this context means that the rule is inserted multiple times according to the specified value.
6. **Storage Period**: This defines how much historical data is retained.

For these problem dimensions, specific values were defined, which were then combined. The values are as follows:

- **Airports**: 1, 10
- **Rule 1**: true, false

- **Rule 2**: true, false
- **Rule Duplication**: 1, 2, 3, 5, 10, and 20
- **Storage Period**: 30, 60, 120

Subsequently, combinations that were deemed to provide little insight were excluded!

6 Mapping Datalog to SQL for knowledge graph representation

This chapter shows avenues of mapping Datalog to equivalent structures in SQL, as an alternative to a native RDF-based representation of the knowledge graph (KG).

6.1 Datalog for knowledge graph representation

A Datalog program consists of facts and rules [42]. Facts represent relevant assertions of the real world [42]. In the context of AWARE, a fact could be "flight *f1* arrives at *Frankfurt Airport*". Rules are sentences that allow a deduction of facts from other facts [42]. In AWARE, a simple rule could be "if a flight departs from airport X and arrives at airport Y, then it connects X and Y". *X* and *Y* in the described rule are variables. In Datalog, facts and rules are expressed in the general shape of $L_0 : -L_1, \dots L_n$ [42]. *L* consists of predicate symbols p_i and terms t_j in the form of $p_i(t_1, \dots, t_j)$, where a term is a constant or variable [42]. The left side of a Datalog clause is called *head* and the right side *body* [42]. If the body of a Datalog clause is empty, the clause is a fact; otherwise, it is a rule [42]. Potential Datalog facts and rules in the setting of air traffic management are shown via the following expressions that describe facts and one rule. For a Datalog program to be valid, facts must not contain variables, and each variable that occurs in a rule head must also occur in its body [42]. These constraints guarantee a finite set of facts that can be derived by the Datalog program [42].

$$departsFrom(f1, ViennaAirport)$$
$$arrivesAt(f1, FrankfurtAirport)$$
$$connects(F, D, A) : -departsFrom(F, D), arrivesAt(F, A)$$

For a Datalog program, two sets of clauses are defined: a set of facts without variables called the Extensional Database (EDB) and a set of rules called the Intensional Database (IDB) [42]. Predicates occurring in the EDB and IDB are disjoint [42]. The EDB is physically stored in a relational database so that each EDB-predicate r corresponds to exactly one relation R in the database, such that each fact $r(c_1, \dots, c_i)$ is stored as tuple $< c_1, \dots, c_n >$ [42]. EDB-predicates are allowed in a Datalog program, but only if they occur in the body of a Datalog rule [42]. For the head only predicates of the IDB are allowed [42]. IDB-predicates can be identified with relations, called derived relations that are defined by the Datalog program and the EDB [42]. Derived relations are not stored explicitly and are considered equivalent to *views* in a relational database [42]. A Datalog program can therefore be considered as a query over the EDB which produces a derived relation [42].

As an example, consider the Datalog program P over an EDB shown in Listing 17. The EDB consists of the two relations *DEPARTS_FROM* and *ARRIVES_AT* with their schemes:

$$DEPARTS\_FROM(FLIGHT, AIRPORT, TIME)$$
$$ARRIVES\_AT(FLIGHT, AIRPORT, TIME)$$

A Datalog program P should now derive the following relation:

$$R_{connected}(Flight, DepartureAirport, ArrivalAirport)$$

This is achieved by using a Datalog rule r_1

$$r_1: R_{connected}(F, D, A) :- DEPARTS\_FROM(F, D, \_), ARRIVES\_AT(F, A, \_)$$

The rule r_1 produces the tuples:

$$< f1, Vienna\ Airport, Frankfurt\ Airport >,$$
$$< f2, Vienna\ Airport, Frankfurt\ Airport >$$

```
DEPARTS_FROM(f1, Vienna Airport, 01.03.2025 15:00)
DEPARTS_FROM(f2, Vienna Airport, 02.03.2025 12:00)
ARRIVES_AT(f1,Frankfurt Airport, 01.03.2025 17:00)
ARRIVES_AT(f2,Frankfurt Airport, 02.03.2025 14:00)
```

Listing 17: Simple Datalog EDB

6.2 Temporal context in Datalog

The use case of AWARE that is covered in this work requires a temporal context dependency for flight data. The temporal context allows for assertions of the data depending on the time it was captured. For example, consider a flight f_1 that is tracked over its entire flight duration and with its flight position captured in a regular time interval. This associated time interval is described by an incrementally increasing *timestep* variable. Consider the EDB of departing and arriving flights shown in Listing 17, but it is extended by a temporal context. If the time of arrival and departure are in the future, they are not considered as certain but represent likely estimates based on external factors. Listing 18 shows the contextualized EDB of the $DEPARTS\_FROM$ and $ARRIVES\_AT$ relations.

```
DEPARTS_FROM(f1, Vienna Airport, 01.03.2025 15:00, 1)
DEPARTS_FROM(f2, Vienna Airport, 02.03.2025 12:00, 1)
ARRIVES_AT(f1,Frankfurt Airport, 01.03.2025 17:00, 1)
ARRIVES_AT(f2,Frankfurt Airport, 02.03.2025 14:00, 1)

DEPARTS_FROM(f1, Vienna Airport, 01.03.2025 15:00, 2)
DEPARTS_FROM(f2, Vienna Airport, 02.03.2025 13:00, 2)
ARRIVES_AT(f1,Frankfurt Airport, 01.03.2025 17:00, 2)
ARRIVES_AT(f2,Hamburg Airport, 02.03.2025 14:00, 2)
```

Listing 18: EDB for a Datalog program with temporal context.

A rule r_2 should now derive a new relation:

$$R_{connected2}(Flight, DepartureAirport, ArrivalAirport, Timestep).$$

The derived relation $R_{connected2}$ should only include tuples that match in their $timestep$. This is achieved by the rule:

$$r_2: R_{connected2}(F, D, A, TS) :- DEPARTS\_FROM(F, D, \_, TS), ARRIVES\_AT(F, A, \_, TS)$$

The rule r_2 produces the tuples:

$$< f1, Vienna\ Airport, Frankfurt\ Airport, 1 >$$
$$< f2, Vienna\ Airport, Frankfurt\ Airport, 1 >$$
$$< f1, Vienna\ Airport, Frankfurt\ Airport, 2 >$$
$$< f2, Vienna\ Airport, Hamburg\ Airport, 2 >$$

When comparing the tuples $< f2, Vienna\ Airport,\ Frankfurt\ Airport,\ 1 >$ and $< f2,\ Vienna\ Airport,\ Hamburg\ Airport,\ 2 >$, notice the change in $ArrivalAirport$ between the first and second timestep.

6.3 Mapping Datalog relations to SQL

The EDB, consisting of the set of ground facts, is physically stored before rules of the IDB derive new relations [42]. The structure of the EDB already follows a relational layout, which allows a straightforward mapping to relational tables in PostgreSQL. Derived relations of the IDB are not explicitly stored. This makes them conceptually equivalent to relational views [42]. Figure 9 shows a mapping from relations of the Datalog EDB and IDB to equivalent SQL tables and views. Tuples of Datalog relations are identical to tuples of SQL relations. Depending on the use case, it may be warranted to map IDB relations to tables or materialized views instead of regular views. For example, if derived data is accessed frequently or deriving facts is costly, materializing derived data in materialized views or tables may be beneficial. Differences between using views, materialized views, or tables are more closely investigated in Chapter 9

Co-funded by the European Union

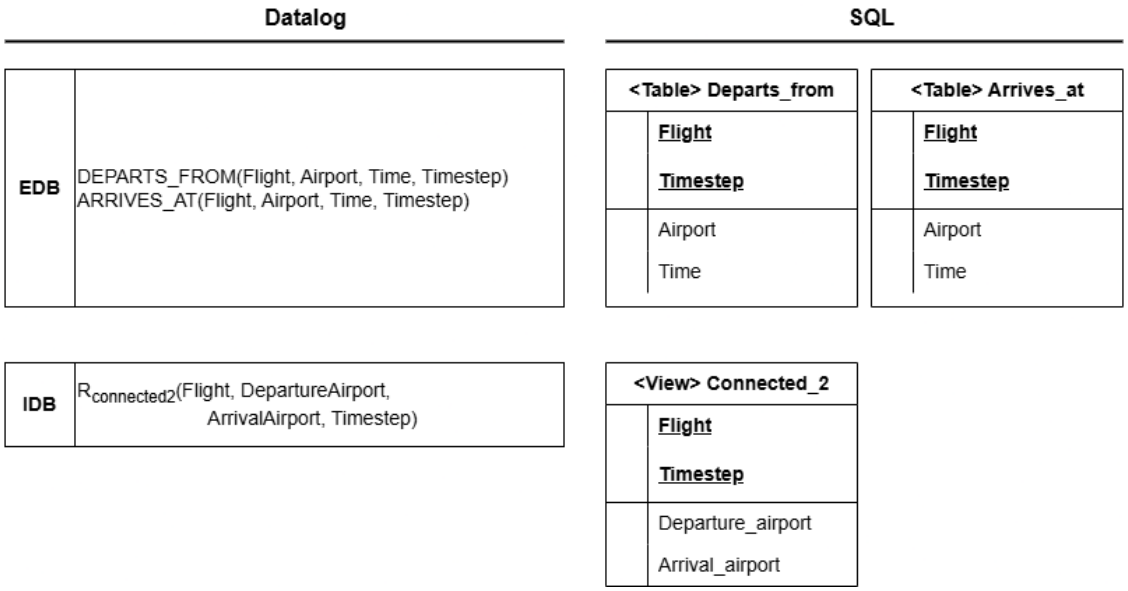


Figure 9: Mapping from Datalog EDB relations to SQL tables and from IDB relations to SQL views.

6.4 Mapping Datalog rules to SQL

Mapping Datalog rules to SQL is dependent on the specific data structure in SQL. Views and materialized views are defined via a *SELECT* statement, whereas the creation of a table follows a specific *CREATE TABLE* syntax. The following comparison between views and tables is based on the EDB defined in Listing 18 and the associated rule r_2.

A Datalog rule is essentially a query over a defined EDB that produces a derived relation. The rule r_2 produces the derived relation $R_{connected2}(F, D, A, TS)$ by querying the relations $DEPARTS\_FROM$ and $ARRIVES\_AT$ of the EDB. A *SELECT* statement is also a query over relations of the database that produces a new relation, commonly based on projection and selection operations. Figure 10 shows the Datalog rule r_2 and its equivalent view definition in SQL. The *CREATE VIEW* statement defines the derived relation $R_{connected2}$, specified in the rule head of r_2, implicitly through the columns projected by the *SELECT* query. The rule body of r_2 is equivalent to the *SELECT* query.

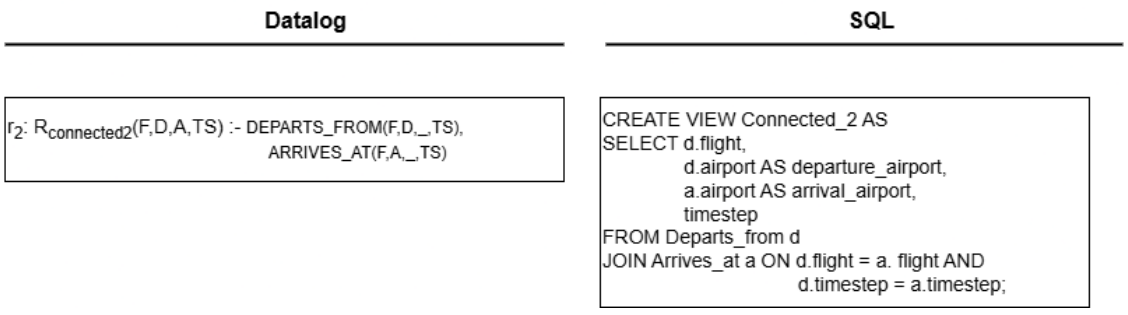


Figure 10: Mapping a Datalog rule to an SQL view definition.

If the derived relation $R_{connected2}(F, D, A, TS)$ is realized as an SQL table, the derived facts are stored explicitly. Contrary to a view, a table cannot be dynamically defined by a *SELECT* query. The Datalog rule is then not directly mapped to a single equivalent SQL data structure, but rule head and rule body are mapped separately to SQL. Consider the EDB of Listing 18 and the goal to map r_2 to SQL by using tables instead of views. First, the relation $R_{connected2}(F, D, A, TS)$ of the rule head is defined, without any associated derivation rules, through a *CREATE TABLE* statement. The derivation rule body is then implemented as an *INSERT INTO* statement, using the same query as was used for the view definition. Figure 11 shows the Datalog rule r_2 and its separation into SQL statements. One advantage of using views instead of tables for rule representation arises when rules are dependent on preceding rules, in terms of scheduling. Consider two rules r_a and r_b, where r_b derives facts based on facts derived by r_a. If both rules are represented as views, the view definition of r_b includes querying r_a. Therefore, if r_b is queried and facts are derived, facts of r_a are automatically derived in advance. If tables would be used instead, a custom scheduling logic must be implemented, for example by using user-defined functions, triggers and external scheduling applications. Developing and maintaining custom scheduling tasks may become impractical for large systems.

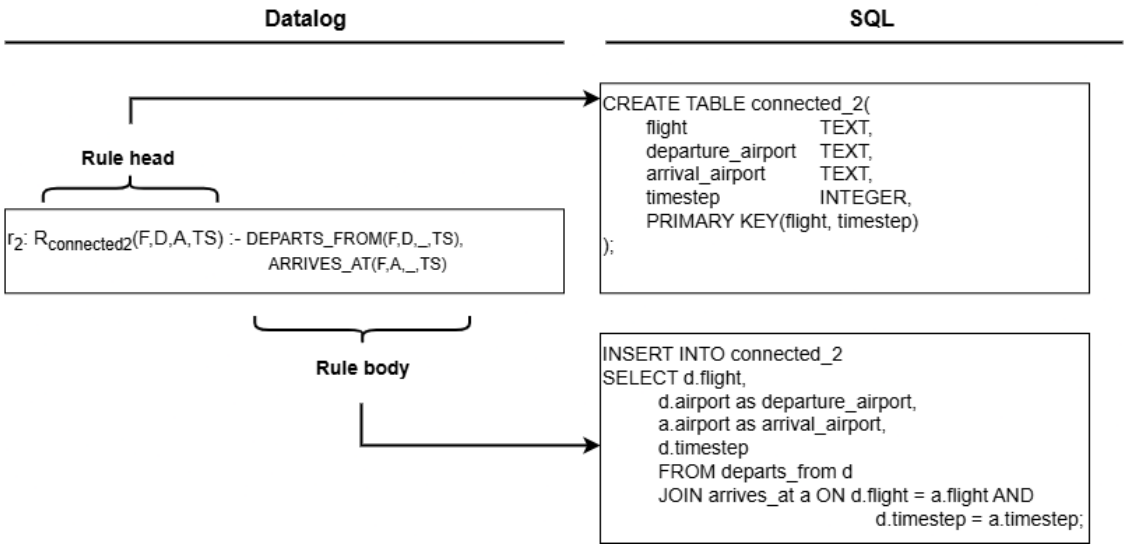


Figure 11: Mapping a Datalog rule to an SQL table

Some Datalog rules may be mapped to SQL tables by using generated columns. A generated column is a special column that is always computed from the values of other columns [43]. When creating a table, an expression used to generate the values of the column is defined in the *CREATE TABLE* statement. This expression can be understood as a simple Datalog derivation rule with additional restrictions. Concrete restrictions, generation expressions, and general functionality of generated columns are DBMS dependent. For PostgreSQL, restrictions for generation expressions include, but are not limited to:

- Used functions must be immutable [43].

- Subqueries are not allowed [43].
- An expression may only reference its current row [43].
- An expression cannot reference other generated columns [43].
- A generated column cannot have a default value [43].

The shown restrictions greatly limit the expressiveness of generated columns. Even though the previously described rule r_2 is rather simple, it cannot be realized by generated columns due to limitations on subqueries and referencing. Consider instead a Datalog rule r_3: $R_{traveledD}(F, D, D_{nm})$:- FLIGHT(F, D), $D_{nm} = D / 1852$ which takes the already traveled distance of a flight in meters and converts it to nautical miles. Mapping r_3 to SQL when using a table for storage of the derived relation $R_{traveledD}(F, D_{nm})$ can be achieved in a similar way as previously shown in Figure 11. The rule head would be mapped to a table, while the rule body would be mapped to an *INSERT INTO* statement. By using a generated column, the derivation of D_{nm}, may be moved from the *INSERT INTO* statement to the *CREATE TABLE* statement. Prerequisite for this approach is that the derived Relation, $R_{traveledD}(F, D, D_{nm})$, includes all terms used to derive D_{nm}. Figure 12 shows how the rule body is separated and mapped to its associated counterparts in SQL, as well as the comparison to a mapping without using a generated column.

Recursive query capabilities were specified for SQL in the *SQL:1999* standard via recursive common table expressions (RCTEs) [44]. Today, most vendors provide RCTE capabilities according to the *SQL:1999* standard [44]. Burzańska et al. [44] provide a comparison of existing RCTE implementations to the *SQL:1999* standard and evaluate the performance of popular relational DBMSs via an RCTE benchmark. RCTEs in PostgreSQL allow for specifications of recursive queries; however, internally those queries are evaluated iteratively [45]. SQL still falls under some restrictions; for example, the *SQL:1999* standard limits capabilities to only linear recursion [44]. However, for the specific use case investigated in this work, recursive capabilities are not required. Therefore, further considerations of recursive performance and capabilities of a relational system are out of scope and remain for future research.

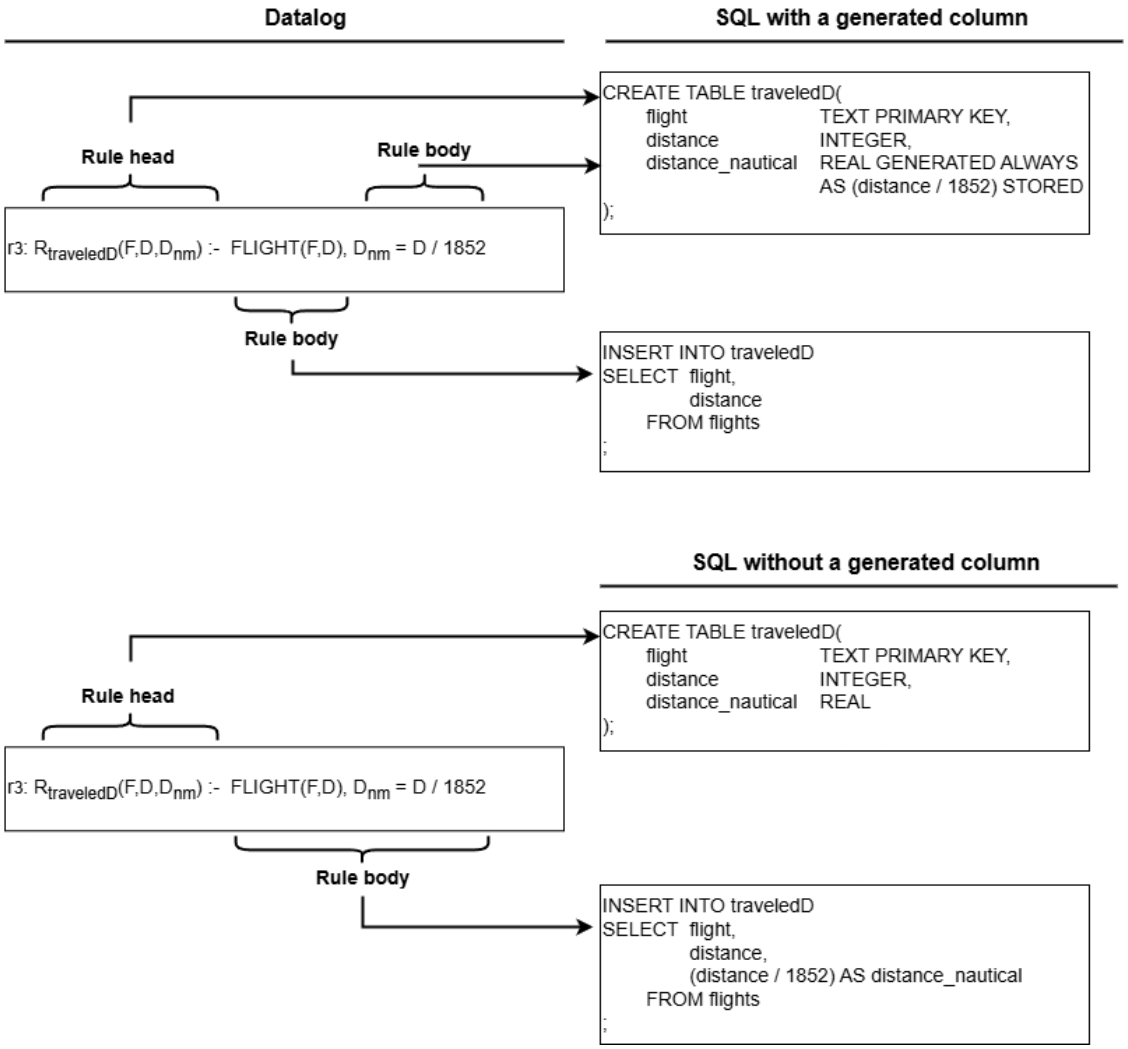


Figure 12: Mapping a Datalog rule to an SQL table by using a generated column.

7 Relational storage layouts for RDF data

In this chapter, the previously described *vertical store*, *horizontal store*, and *type stores* are investigated for their suitability for air traffic management. For comparing the storage approaches, a directed edge-labelled graph, shown in Figure 13, is used as the running example.

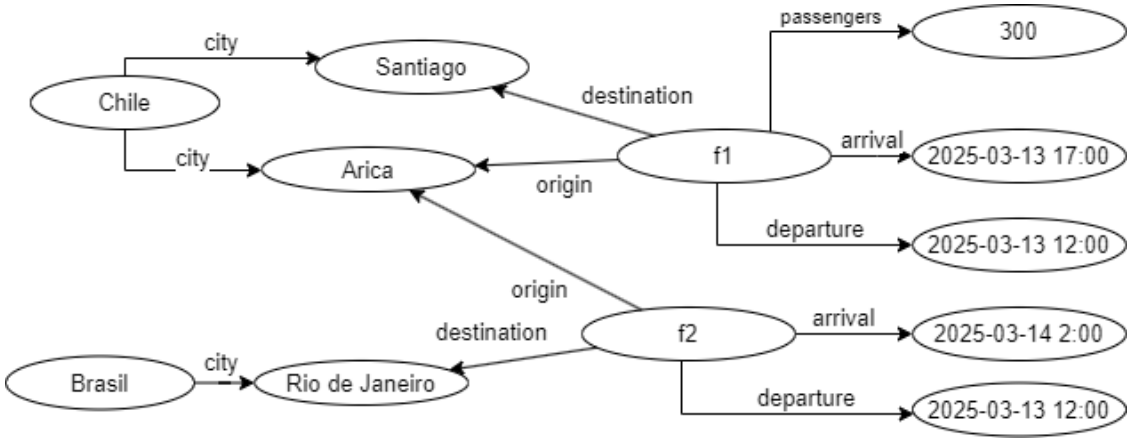


Figure 13: A directed edge labeled RDF graph showing data of two flights.

7.1 Vertical store

A vertical store maps the triples of an RDF KG directly to a fixed relational schema and disregards a consideration of an advanced relational schema [32]. However, a vertical store may still be practical for certain use cases. First, a context-independent mapping of an RDF KG to a vertical store is described in Section 7.1.1, and then context-dependency of the KG is recognized in Section 7.1.2.

7.1.1 General vertical store

A vertical store, or triple store, uses a fixed relational schema in the form of $< subject, predicate, object >$ [32]. This has the major advantage that a vertical store can handle any dynamic RDF schema [32]. The underlying RDF graph may grow continuously, but it can always be stored in the vertical store without having to adapt the schema. Converting the RDF graph shown in Figure 13 to a vertical relational table results in Table 4. If the graph were to be expanded by two edges capturing the continent of Chile and Rio de Janeiro, two new tuples would be added to the vertical store without requiring modification of the data schema. Another advantage is the flexibility concerning missing attributes of entities. The edge $f1 - passengers - 300$ captures the number of passengers on the flight. Flight $f2$ could be a private flight, leading to unavailable passenger information and therefore to a missing $f2 - passenger - count$ edge. As each triple in the RDF graph maps to one tuple in the vertical store, the tuple $< f2, passengers, count >$ is also missing in the vertical store.

Subject	Predicate	Object
f1	arrival	13.03.2025 17:00
f1	departure	13.03.2025 12:00
f1	origin	Arica
f1	destination	Santiago
f1	passengers	300
f2	arrival	13.04.2025 02:00
f2	departure	13.03.2025 12:00
f2	origin	Arica
f2	destination	Rio de Janeiro
Chile	city	Santiago
Chile	city	Arica
Brasil	city	Rio de Janeiro

Table 4: Vertical store for RDF data

The approach of disregarding advanced schemas of relations and only mapping to a generic triple relation comes with the disadvantage of bypassing optimization techniques of the DBMS, which would otherwise be available out of the box. Querying data is reliant on self-joins of the $subject - predicate - object$ relation and will quickly encounter scalability issues when not using specialized techniques [32]. For example, an excessive use of indexing may improve self-join performance [32]. Consider the previously proposed rule r_1 in Section 6.1 that derives a relation of flights with their origin and destination airports. A rule $r_1 *$ should now derive a similar relation $R(F, O, D)$ based on the predicates $origin$ and $destination$ in the vertical store. Instead of joining separate relations, the vertical store is self-joined to retrieve subject and object values of tuples with "origin" and "destination" as predicates. Figure 14 shows the difference between using Datalog and SQL when the data of Table 4 is provided as a database.

Datalog

```
r1*: Rconnected(F,O,D) :-  vertical_store(F, "origin", O),
                           vertical_store(F, "destination", D).
```

SQL

```
CREATE VIEW connected AS
SELECT
   orig.subject AS F,
   orig.object AS O,
   dest.object AS D
FROM
   vertical_store orig
JOIN
   vertical_store dest ON orig.subject = dest.subject
WHERE
   orig.predicate = 'origin'
   AND dest.predicate = 'destination'
;
```

Figure 14: Querying a vertical storage layout with Datalog and SQL. The derived relation in SQL is captured as a view.

Consider now an extension of rule $r_1 *$, rule $r_1 +$, which additionally derives the departure and arrival time of flights as relation $R(F, O, D, DepT, ArrT)$. Each additional variable in the derived relation requires an additional self-join of the source relation. In comparison, when considering relations $DEPARTS\_FROM(FLIGHT, AIRPORT, TIME)$ and $ARRIVES\_AT(FLIGHT, AIRPORT, TIME)$ as proposed in Section 6.1, the extension of $r_1 *$ to $r_1 +$ would not cause any additional joins. The proportionally increasing number of self-joins with an increasing number of derived variables poses a tradeoff between querying performance/complexity and schema flexibility for the vertical store.

7.1.2 Contextualized vertical store

The vertical store in the form of $R(subject, predicate, object)$ poses a straightforward mapping for an RDF graph with similar edges $subject - predicate - object$. Each triple in a graph is directly mapped to a tuple of the relation. Extending RDF with context dimensions is not trivial and may follow various approaches, as highlighted in Section 3.1. Providing context to tuples in a vertical store faces similar challenges. In this section, three different approaches to contextualize a vertical store are described in the domain of air traffic management. The approaches are further referred to as *tuple-level context*, *context relations*, and *horizontally split contextualization*.

Context may be provided to RDF graphs by using reification [5]. When reifying, edges are seen as separate entities with associated edges $subject$, $predicate$ and $object$ [5]. Reified edges directly represent the schema of the investigated vertical store. To provide context to reified edges in an RDF graph, additional edges representing the context are added to the reified edge [5]. This can be mapped to the vertical store by adding the context edges as additional attributes for each tuple. Figure 15 shows a change of the arrival time for flight $f1$ and its associated temporal context. The temporal context can be directly added to the tuples; *NULL* values are assigned where the context is not available. This is considered *tuple-level context*, as for each tuple individually the context is added, if available.

Tuple-level context is based on the general concept of *Type 2* slowly changing dimensions (SCDs) in data warehousing [46]. The authors propose the addition of at least 3 columns to capture the time interval of row validity and an indicator to denote the current valid row in the *Type 2* SCD table [46].

Co-funded by the European Union

In data warehousing, the concept of a slowly changing dimension would, for example, be applied to a dimension representing departments of a company [46]. Tracking changes of context, when, for example, the department changes its location, is essential to enable analytics over past data [46]. However, when tracking flights in air traffic management and, more specifically, in AWARE, context changes happen with a much higher frequency of up to once per second. The two attributes needed to capture the valid interval of a row are considered too excessive for the derivation rules investigated in this work. The authors also introduce a surrogate key in the context dimension, which simplifies referencing multiple context dimensions in a single fact table [46]. Using a surrogate key as a primary key, instead of a natural key and its context, makes the system resistant against changes of business rules and to design differences when sourcing from multiple systems [46]. For the use case in this work, using a surrogate key instead of a combination of a natural key and context attributes is possible but disregarded for the regular *tuple-based* approach. The reason behind this decision is that for the investigated specific AWARE rules in this work, only a single context dimension is used. However, a surrogate key is used in the approaches of *context relations* and *horizontally split contextualization*.

In general, the primary key of the vertical store table must include all available context dimensions to uphold the ability to uniquely identify triples over multiple contexts, despite duplicate subjects. Table 5 shows the resulting vertical store table when the $valid\_from$ context is only added to the $arrival$ edge of flight $f1$. Adding new context types to edges requires an update of the data schema of the vertical store. If context is only available for a small number of tuples, or many different context attributes are required, this approach may result in a high number of *NULL* values.

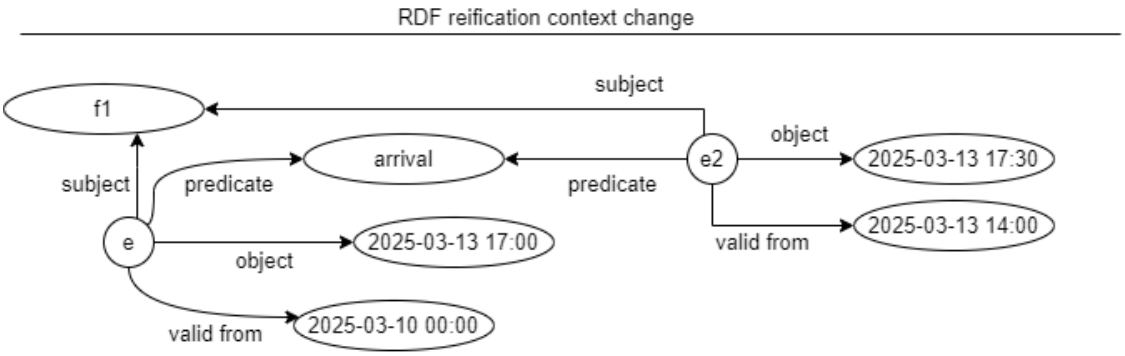


Figure 15: Change of temporal context of a reified edge.

Subject	Predicate	Object	valid_from
f1	arrival	13.03.2025 17:00	2025-03-10 00:00
f1	arrival	13.03.2025 17:30	2025-03-13 14:00
f1	departure	13.03.2025 12:00	NULL
f1	origin	Arica	NULL
f1	destination	Santiago	NULL
f1	passengers	300	NULL
f2	arrival	13.04.2025 02:00	NULL
f2	departure	13.03.2025 12:00	NULL
f2	origin	Arica	NULL
f2	destination	Rio de Janeiro	NULL
Chile	city	Santiago	NULL
Chile	city	Arica	NULL
Brasil	city	Rio de Janeiro	NULL

Table 5: Vertical store for RDF data with added context valid_from

When using a *context relation*, the goal is to mitigate the problem of having to update the schema of the vertical store table repeatedly due to the introduction of additional contexts and the problem of creating a sparsely populated table when required contexts vary for tuples. Consider a relation $Rc(cid, c_1, \ldots, c_n)$ that holds multiple contexts $(c_1, \ldots, c_n)$ and assigns for each unique combination of contexts a unique key cid. Instead of providing each context individually for each tuple in the vertical store table, tuples reference cid of the context relation instead. Introduction of additional context dimensions then only affects the context relation. When a new context combination is created, tuples affected by a context change are inserted into the vertical store table with the associated context id. This design also prevents required updates to the primary key of the vertical store table each time an additional context attribute is added. Table 6 shows an example context relation Rc with a temporal $valid\_from$ and a provenance $sourced\_from$ context. Table 6 shows an exemplary corresponding vertical store table.

cid	valid_from	sourced_from
1	13.03.2025 17:00	NULL
2	13.03.2025 17:30	www.example.com
3	NULL	www.example.com
4	NULL	NULL

Table 6: Context relation with a temporal valid_from and a provenance sourced_from context.

Subject	Predicate	Object	cid
f1	arrival	13.03.2025 17:00	1
f1	arrival	13.03.2025 17:30	2
f1	departure	13.03.2025 12:00	4
f1	origin	Arica	4
f1	destination	Santiago	4
f1	passengers	300	4
f2	arrival	13.04.2025 02:00	4
f2	departure	13.03.2025 12:00	4
f2	origin	Arica	4
f2	destination	Rio de Janeiro	4
Chile	city	Santiago	3
Chile	city	Arica	3
Brasil	city	Rio de Janeiro	3

Table 7: Vertical store for RDF data with added context by referencing a context relation.

Both previously investigated approaches use only a single table for all available base data and derived data. For large graphs, or when context changes frequently, the data set might grow too large to efficiently query. Consider, for example, a knowledge graph representing the current positions of aircraft. The knowledge graph is used to visualize flight positions on a screen in near real-time. For each update cycle, the position of all aircraft would change simultaneously. Given a temporal context in the form of a single $valid\_from$ attribute, all aircraft positions are inserted again into the vertical store table. An aircraft position can be described via $latitude$, $longitude$ and $altitude$, which results in a growth of the data set by $(3 \times \#Aircraft)$ tuples per update cycle. To keep the size manageable

and querying performance high, it may be beneficial to split the data set horizontally according to the context. For each update cycle a new table is created, which contains all data associated with the current temporal context. Additionally, one tuple in the context relation is created, which maps context to a unique cid and, by extension, indicates the availability of a vertical store table associated with that context. This approach is based on the idea of contextualizing RDF graphs by using *named graphs*. A major disadvantage of this approach is the vastly increased complexity for queries and the schema. The retrieved cid of a context must be mapped to the name of a vertical store table that holds the contextualized data. This dynamic mapping is not supported by standard SQL and must be performed by an external client or by a function written in a procedural language in the DBMS. For example, in PostgreSQL the mapping may be realized as a user-defined PL/pgsql function and is shown in Figure 16.

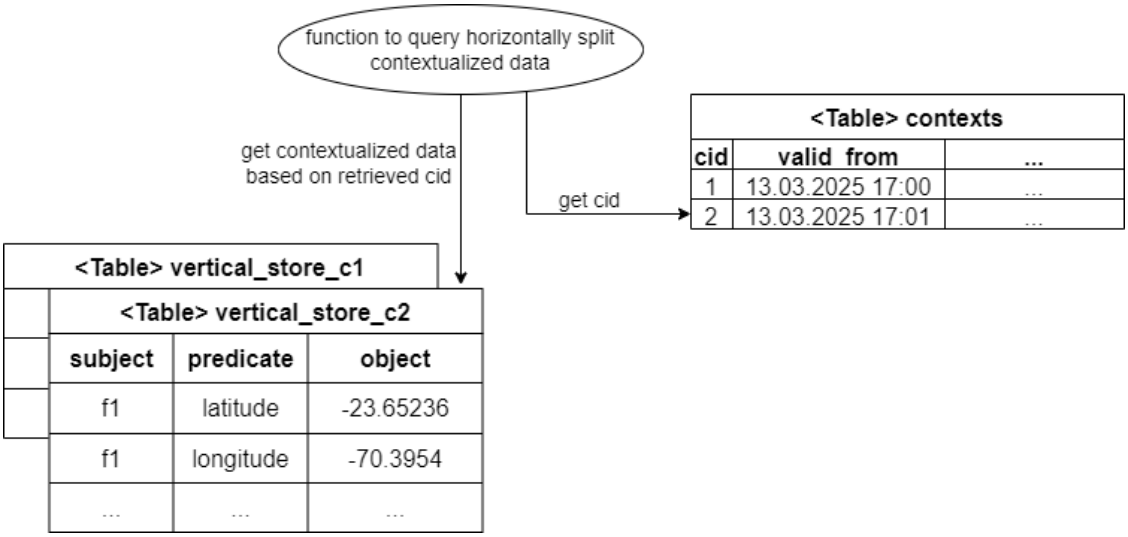


Figure 16: Querying horizontal splits of a contextualized, vertical store.

7.2 Horizontal store

Using a horizontal representation, RDF data can be stored directly in a single table [32]. Instead of reducing the schema to a generic form of $subject - predicate - object$ as in the vertical store, the schema of the horizontal store table is continuously expanded to accommodate all occurring edges in the graph. The table of the horizontal store has one column for each predicate and one row for each subject [32]. Given an RDF triple in the form of $(s - p - o)$, the object o is in row s and column p [32]. If the subject and predicate of two triples are equal, o is mapped to the same cell for both triples [32]. This results in a set of values $\{o_1, \ldots, o_n\}$ in a single cell [32]. If subjects do not fully share the same set of predicates, some cells will map to non-existent triples of the graph; in this case, cells are assigned *NULL* values [32]. The table resulting from the proposed horizontal store approach is similar to the approach of using *one big table* in data warehousing.

Given the RDF graph shown in Figure 13, a conversion to a horizontal store table would result in Table 8. Even though the graph is rather simple, the flexible schema of RDF already results in a sparsely populated table. Additionally, the triples $(Chile - city - Arica), (Chile - city - Santiago)$ produce a set of values for a single cell. Certain subjects in an RDF graph are prone to have a high number of different object values for the same predicate. For example, in air traffic management, a

single country will have many relevant airports that could be captured by similar predicates. This would produce a large set of values in a single cell, making the data not only difficult to further work with but also violating the first normal form of the relation.

Subject	Arrival	Departure	Origin	Destination	Passengers	City
f1	13.03.2025 17:00	13.03.2025 12:00	Arica	Santiago	300	NULL
f2	13.04.2025 02:00	13.03.2025 12:00	Arica	Rio de Janeiro	NULL	NULL
Chile	NULL	NULL	NULL	NULL	NULL	{Arica; Santiago}
Brasil	NULL	NULL	NULL	NULL	NULL	Rio de Janeiro

Table 8: Horizontal store for RDF data.

To solve those problems, approaches have been proposed to split the table vertically into sets of smaller tables [32]. The naive approach is to split the table for each predicate, which produces relations in the form of $R(subject, object)$ [32]. Each tuple in the relation represents the binary relation between $subject$ and $object$ with respect to a $predicate$ [32]. If triples of the RDF graph differ in their predicate, they are mapped to different tables [32]. This results in multiple tables, the number of tables being equal to the number of different predicates in the RDF graph [32]. Given the relation of the horizontal store shown in Table 8, splitting as proposed would result in a set of relations shown in Figure 17. Splitting the original horizontal store table into binary relations completely solves the problems of the sparsely populated table and the sets of values [32]. However, this approach comes with the disadvantage of increasing the number of required joins for querying [32]. In terms of schema flexibility, adding new predicates to the RDF graph would result in the creation of new tables.

Arrival relation	
Subject	**Arrival**
f1	13.03.2025 17:00
f2	13.04.2025 02:00

Departure relation	
Subject	**Departure**
f1	13.03.2025 12:00
f2	13.03.2025 12:00

Destination relation	
Subject	**Destination**
f1	Santiago
f2	Rio de Janeiro

Origin relation	
Subject	**Origin**
f1	Arica
f2	Arica

City relation	
Subject	**City**
Chile	Arica
Chile	Santiago
Brasil	Rio de Janeiro

Passengers relation	
Subject	**Passengers**
f1	300

Figure 17: Splitting a horizontal store table into binary relations between subject and object.

7.3 Type store

Using one big table as a horizontal store comes with drawbacks that can be alleviated by using a *type store*. Section 7.3.1 shows a context-independent type store approach, while Section 7.3.2 recognizes context dependency.

7.3.1 General type store

As described in the previous section, using a single horizontal table results in a sparsely populated table with sets of values, while splitting for each predicate results in a high number of joins [32]. Type stores can be seen as a mix between the two horizontal store approaches. The idea is to split the wide horizontal store table into multiple smaller tables, but instead of splitting for each predicate, related predicates are identified and kept in a single table [32]. This approach reduces the number of joins when compared to the fully split horizontal store approach and reduces the number of *NULL* values when compared to the wide horizontal store approach [32]. The problem of sets of values for certain cells is not solved in this approach [32]. Identifying related predicates in an RDF graph and deciding on splitting conditions is not trivial and may use, for example, frequent access patterns or association rule mining [32]. If the structure of the RDF knowledge graph is confined within a domain, domain experts may provide valuable domain knowledge used to decide on specific splits. Given the RDF graph shown in Figure 13, two types of subjects can be identified. The subjects $f1$ and $f2$ represent flights, while $Chile$ and $Brasil$ represent countries. Flights and countries may also be understood as classes, in which case the rdf:type predicate of the RDF schema may be used as an indicator. Assume domain experts affirmed the two classes $flight$ and $country$, then the two relations shown in Figure 18 would

be created. The relations can be further normalized, but depending on the use case, it may be beneficial to keep relations denormalized to decrease required joins and querying complexity. In the $country$ relation, a value set occurs for the tuple $< Chile, \{Arica, Santiago\} >$. The value set may be eliminated by normalization and bringing the relation into the first normal form. By including $City$ in the primary key of the relation and splitting the value set into multiple rows, the value sets are resolved. Another alternative representation to deal with value sets is to use nested tables or array values whenever a relation contains a one-to-many relationship. The relation $R_{country}(subject, City)$ is then represented as $R_{country}(subject, City[])$ with $City[]$ as a variable-length array, or nested table.

Flight relation					
Subject	**Departure**	**Arrival**	**Origin**	**Destination**	**Passengers**
f1	13.03.2025 12:00	13.03.2025 17:00	Arica	Santiago	300
f2	13.03.2025 12:00	13.04.2025 02:00	Arica	Rio de Janeiro	NULL

Country relation	
Subject	**City**
Chile	{Arica; Santiago}
Brasil	Rio de Janeiro

Figure 18: Splitting a horizontal store table into two smaller tables, based on domain knowledge.

7.3.2 Contextualized horizontal stores and contextualized type store

A type store can be considered a special case, or refinement, of a horizontal store. Therefore, challenges of contextualization and approaches to solve them are applicable to both storage types. Two approaches were identified to introduce context to a horizontal store. The first approach is referred to as *tuple-level context*, and the second as *context-dependent relations*. Similar to the *tuple-level context* for the *vertical store*, the *tuple-level context* approach for a horizontal store is based on the same initial idea of SCDs in data warehousing, which was described in Section 7.1.2.

The RDF graph in Figure 19 shows the previously investigated graph of Figure 13, with added contexts to the arrival time of flight $f1$. The graph captures a change in the arrival time of flight $f1$ and the temporal validity of the old and new times. Consider a mapping of the contextualized graph to a horizontal store in the same way as it was previously performed for the non-contextualized variant. The context attribute *valid_from* is added as a column to the table. Context dependency for individual tuples is captured by values in context columns, hence the name *tuple-level context*. The two datetime nodes, $(13.03.2025\ 17{:}\,00)$ and $(13.03.2025\ 17{:}\,30)$, that capture the arrival times for flight $f1$ are considered subjects and are inserted as separate rows into the table. The resulting table is shown as Table 9. This naive approach is infeasible in various aspects. To certainly know whether the set of values for $Arrival$ of $f1$ has an associated context or not, all other subjects would have to be checked. Furthermore, the values for the arrival time $(13.03.2025\ 17{:}\,00; 13.03.2025\ 17{:}\,30)$ are now each unique identifiers for tuples. This is wrong, as in reality the dates are realized as literals and are not valid as a primary key. Theoretically, the date literals could be used as identifiers, but they are not necessarily unique over the whole relation. Consider a flight $f3$ with a value for $arrival$ denoted as $f3\_arr$ and a value for $arrival$ for subject $f1$ denoted as $f1\_arr$. The value $f3\_arr$ is dependent on a temporal *valid_from* context $f3\_arr_c$, while $f1\_arr$ is dependent on $f1\_arr_c$. If $f1\_arr_c <>$

$f3\_arr_c$ and $f1\_arr = f3\_arr$, two tuples are added to the table. Both tuples share the same primary key but differ in the value of $valid\_from$. In the relational representation, the dependency between $arrival$ and $valid\_from$ becomes ambiguous for triple-subjects with equal values for their objects but different context dependencies of those objects. The introduction of a surrogate key as the primary key solves the problem of the primary key violation. However, it does not solve the ambiguous context dependencies unless the original subject references the surrogate key instead of the value of the attribute. The referencing of the surrogate key does only work if it can be ensured that the value of the surrogate key lies outside of the value range for the original attribute. Table 10 shows a problematic representation of a valid knowledge graph to a relational horizontal store table.

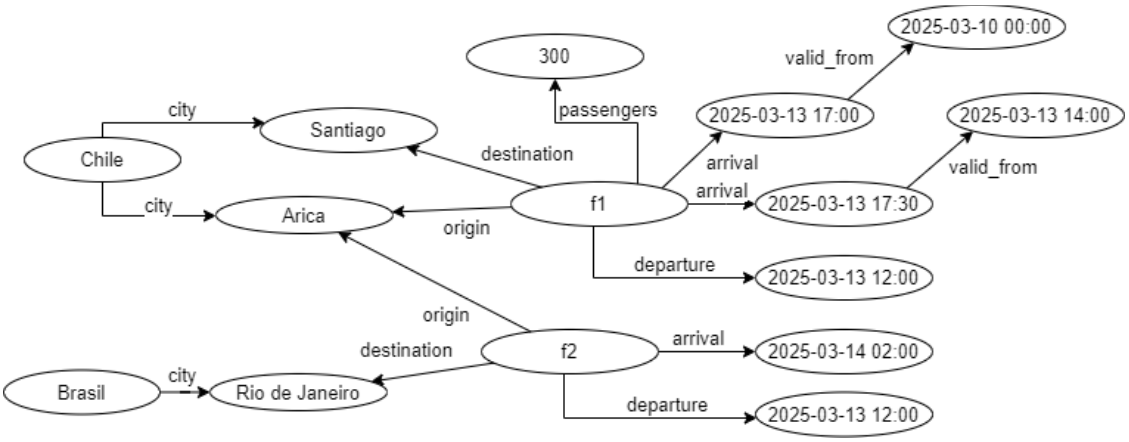


Figure 19: RDF graph with explicit direct context representation for the arrival times of flight f1.

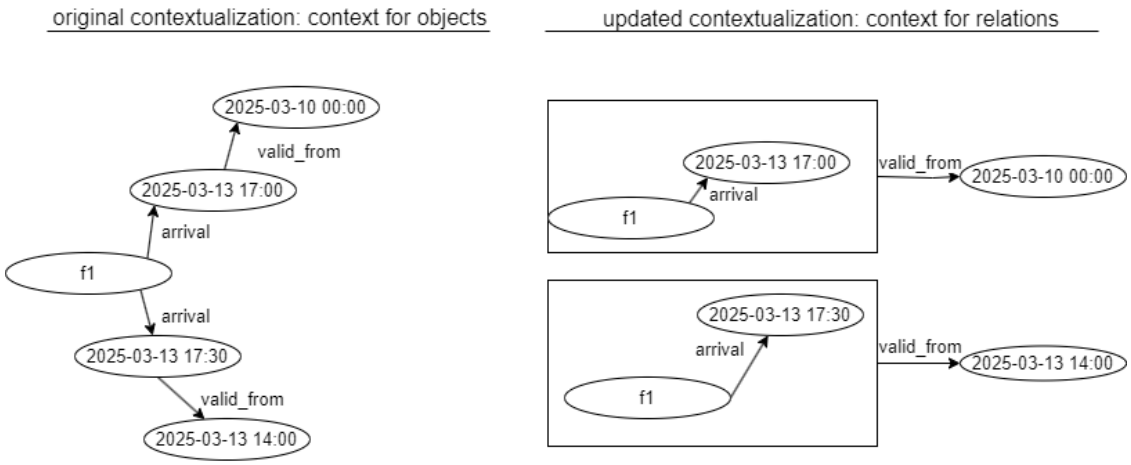
Subject	Arrival	Departure	Origin	Dest.	Pass	City	Valid From
f1	{13.03.2025 17:00; 13.03.2025 17:30}	13.03.2025 12:00	Arica	Santiago	300	NULL	NULL
f2	13.04.2025 02:00	13.03.2025 12:00	Arica	Rio de Janeiro	NULL	NULL	NULL
Chile	NULL	NULL	NULL	NULL	NULL	{Arica; Santiago}	NULL
Brasil	NULL	NULL	NULL	NULL	NULL	Rio de Janeiro	NULL
13.03.2025 17:00	NULL	NULL	NULL	NULL	NULL	NULL	2025-03-10 00:00
13.03.2025 17:30	NULL	NULL	NULL	NULL	NULL	NULL	2025-03-13 14:00

Table 9: Infeasible contextualized horizontal store for RDF data, due to a naive mapping approach

Subject	Arrival	Departure	Origin	Dest.	Pass	City	Valid From
f1	{13.03.2025 17:00; 13.03.2025 17:30}	13.03.2025 12:00	Arica	Santiago	300	NULL	NULL
f2	13.04.2025 02:00	13.03.2025 12:00	Arica	Rio de Janeiro	NULL	NULL	NULL
Chile	NULL	NULL	NULL	NULL	NULL	{Arica; Santiago}	NULL
Brasil	NULL	NULL	NULL	NULL	NULL	Rio de Janeiro	NULL
13.03.2025 17:00	NULL	NULL	NULL	NULL	NULL	NULL	2025-03-10 00:00
13.03.2025 17:30	NULL	NULL	NULL	NULL	NULL	NULL	2025-03-13 14:00
13.03.2025 17:30	NULL	NULL	NULL	NULL	NULL	NULL	2025-03-13 15:30

Table 10: Violation of primary key and ambiguous context dependencies for a contextualized horizontal store

A solution to this problem is to assign a given context c only to a tuple in a relation with the schema $R_c(s, o_1, \ldots, o_n)$, if each value of attributes $(o_1, \ldots, o_n)$ of the tuple is dependent on the context c. This approach is here referred to as *context-dependent relations*. For the mapping, the context can then be treated as object o_c to the subject s in R_c, even though it would in fact be a related object to each other object in R_c. Additionally, each context attribute in R_c is added to the primary key of the relation to enable reasoning over context changes. Splitting a wide relation into smaller relations that contain only related predicates is what differentiates a *type store* from the simpler horizontal store. Therefore, converting a wide horizontal store table into multiple smaller type store tables is a prerequisite to introduce context on *context-dependent relations*. Figure 20 shows the original representation of context in an RDF graph via direct representation and an updated representation. A direct representation of context for triple objects is the variant resulting in the infeasible relation that is shown in the lower part of Figure 20. The updated context representation shows the construction of a relation $flight$ in a *type store* and its equivalent representation as an RDF graph. The collection of multiple edges into a single structure and its contextualization can, for example, be achieved by using *named graphs* [5]. Conceptually, this idea is also strongly related to the representation of context as boxes [47]. Metaphorical boxes are introduced that hold arbitrary representations of content, but the content in the boxes is, at least partially, dependent on additional context information assigned to the boxes themselves [47]. In the example shown in Figure 20, the only box parameter would be the *valid_from* attribute of the flight relation. Compared to the basic idea of context as a box, for the representation of context in this work, content inside boxes must be dependent overall, not a partial, context.



relational representation

Flight relation			
Subject	**valid_from**	**Arrival**	...
f1	2025-03-10 00:00	2025-03-13 17:00	...
f1	2025-03-13 14:00	2025-03-13 17:30	...
...	...	...	...

Figure 20: The RDF graph with explicit direct context representation represents the originally investigated approach to provide context individually for context-dependent object values of triples. This approach is infeasible, which results in an updated variant which results in an updated variant where relations are adapted, so that each object value in a tuple depends on the context.

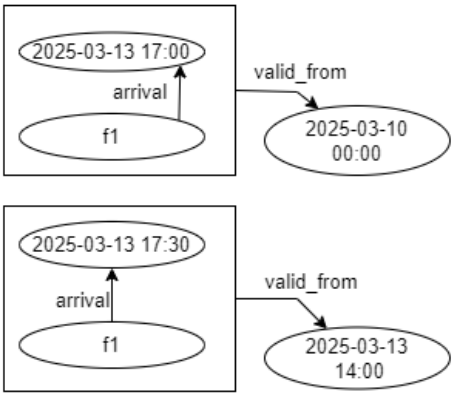
Homola et al. [48] introduce a framework called contextualized knowledge repositories (CKR) to expand on the idea of context as a box by additionally using hierarchical context dimensions, meta knowledge, and context classes. A primitive context has its context boundaries defined via constants, whereas a context class allows for a more generic approach [48]. Generally, context classes serve the purpose of making knowledge representation more effective, as certain axioms that are valid in multiple contexts are automatically imported into all contexts that instantiate the class [48]. Instead of constants for dimensional values, a context class uses concepts that are associated with all primitive contexts that instantiate the class context [48]. CKR contexts are organized in a hierarchical *coverage* relation to regulate propagation of knowledge between them [23]. A context is qualified by a set of dimensional attributes that specify its boundaries [23]. A context C_A with context dimensions A covers a context C_B with context dimensions B if all dimensional values of A cover all dimensional values of B and $B \subseteq A$ [23].

In this work, investigated rules are only dependent on a single temporal context dimension with one level. Therefore, mapping the full capabilities of CKRs to a relational setting is considered out of scope. However, hierarchical context structures may still become relevant in the future. For example, when reasoning over specific time intervals, aggregating exact points of time to higher context levels of the

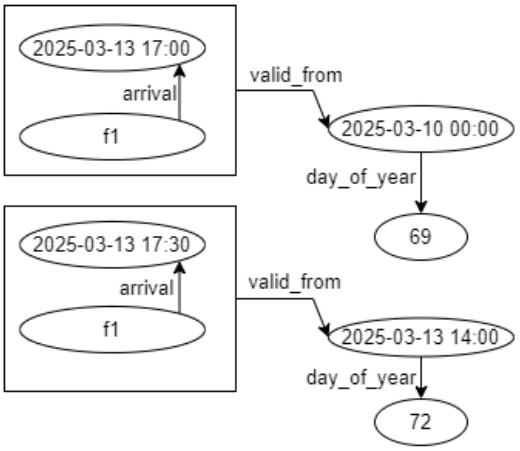
time dimension may be required. Consider aggregating contexts, including a *valid from* attribute, hierarchically to days in a year. Even though a query may extract the day out of the date attribute and convert it to its day_of_year value to use it for aggregation, providing the day_of_year as an attribute beforehand may be beneficial for performance. The proposed dimensional structure follows the idea of a dimensional fact model in data warehousing. The flight relation $R_{flight}(subject, valid\_from, arrival)$ has one associated context attribute $valid\_from$. The goal is to include a context attribute, day_of_year, which represents a higher-level temporal context. Both context attributes are part of the same context dimension C_t. The day_of_year context attribute could be included directly into the relation R_{flight}, but instead, a dimensional relation with schema $R_{time}(id, valid\_from, day\_of\_year)$ is introduced. The original flight relation then references the surrogate key (id) in R_{time}, which corresponds to the originally assigned $valid\_time$. A day_of_year dimensional value covers a $valid\_from$ value if the date representation of $valid\_from$ can be converted to the same day_of_year value. This schema layout follows the idea of a star schema in data warehousing [46]. The base relation R_{flight} is treated like a fact table, and context relations are treated as dimension tables. Figure 21 shows an expansion of the simple context $valid\_time$ in Figure 20 by introducing an additional context level for day_of_year.

This approach is also applicable for multidimensional context dependencies. Each context dimension is captured as a relational table and is referenced by a surrogate key in context-dependent relations. Assume flights are dependent on two different context dimensions, the previously described temporal time context C_t and a provenance context describing the source of the data C_s. For example, the context dimension of C_s spans two levels, $sourced\_from$ and $type$. The dimensional value $'external\ module'$ for $type$ covers the dimensional values $('flight\ schedule\ module'$ V $'delay\ prediction\ module')'$ for $sourced\_from$. Consider the external module *flight schedule module* that calculates the expected arrival time of a flight when it is first scheduled. The flight $f1$ is planned by the *flight schedule module*, and its data is captured in the relation R_{flight}. During transit of the $f1$ external factors, like weather changes, cause a previously unexpected delay that is predicted by a *delay prediction module*. A new tuple is inserted into relation R_{flight}, which is dependent on both context dimensions and references the associated dimensional values in the respective context relations via a surrogate key.

single-level context



introduction of an additional context level



relational representation of a single-level context

Flight relation			
Subject	**valid_from**	**Arrival**	...
f1	2025-03-10 00:00	2025-03-13 17:00	...
f1	2025-03-13 14:00	2025-03-13 17:30	...
...	...	...	...

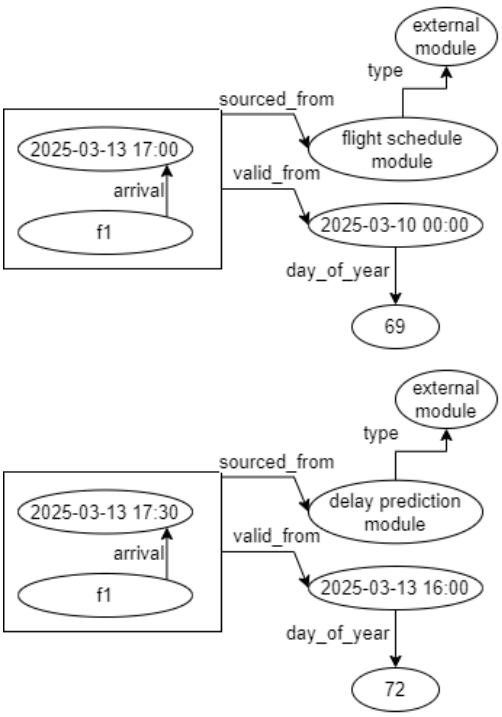
relational representation

Flight relation			
Subject	**time**	**Arrival**	...
f1	1	2025-03-13 17:00	...
f1	2	2025-03-13 17:30	...
...	...	...	...

Time relation			
id	**valid_from**	**day_of_year**	...
1	2025-03-10 00:00	69	...
2	2025-03-13 14:00	72	...
...	...	...	...

Figure 21: The left-hand side shows the representation of context for relations so that each attribute is dependent on the context. This approach was previously already shown in Figure 20. The right-hand side transforms the time context to a multilevel hierarchical context by introducing day_of_year as a generalized level to $valid\_from$, similar to a dimension table in a star schema used in relational data warehousing.

multidimensional and multilevel contexts



relational representation of multidimensional and multilevel contexts

Flight relation

Subject	time	source	Arrival	...
f1	1	1	2025-03-13 17:00	...
f1	2	2	2025-03-13 17:30	...
...	...		...	...

Time relation

id	valid_from	day_of_year	...
1	2025-03-10 00:00	69	...
2	2025-03-13 16:00	72	...
...	...	...	...

Source relation

id	sourced_from	type	...
1	flight schedule module	external module	...
2	delay prediction module	external module	...
...	...	...	...

Figure 22: An additional context dimension is added to the knowledge graph and its relational representation. Further context dimensions may be added via this approach, under the prerequisite that each attribute in the base relation is dependent on the context.

7.4 Choosing a contextualized storage layout

The previous sections discussed general aspects about contextualized *vertical stores*, *horizontal stores*, and contextualized *type stores*. In this section, advantages and disadvantages of the investigated storage layouts are briefly compared with respect to the requirements of the AWARE project, and the most promising storage layout is applied to the specific use case.

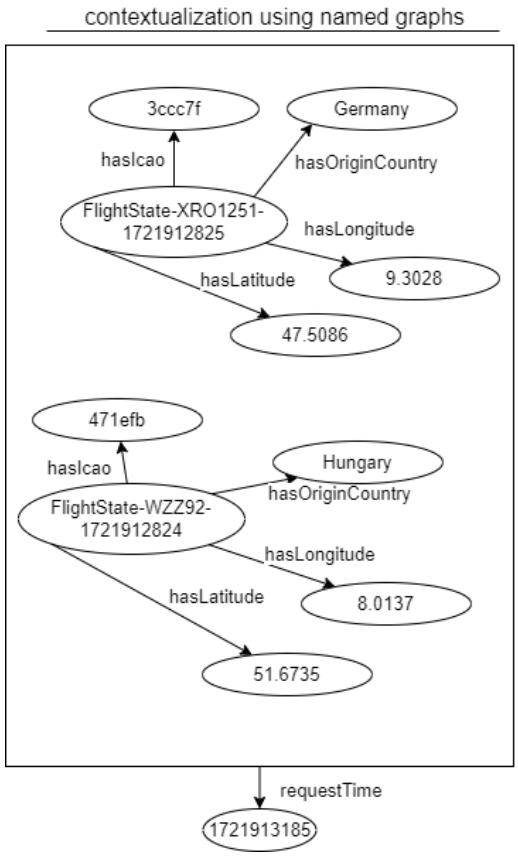
A *vertical store* with a relational schema of $R(subject - predicate - object)$, or in the case of a contextualized variant, in the relational schema of $R(subject, predicate, object, c_1, \ldots, c_n$ has the major advantage of schema flexibility. Each edge in an RDF graph is reified and can be mapped to the same schema. When edges are added to the RDF graph with new predicates, those edges are added as new tuples to the relation. Consider a newly implemented module that provides weather data. For example, a new edge $(w1 - weatherCondition - normal)$ would map to a tuple $< w1 - weatherCondition - normal >$ without any update to the relational schema. This flexibility may benefit a fast, exploratory development. For the exploratory research in AWARE, the schema is well defined, and the data is structured. The KG in AISA was mostly mapped from exchange models defined in UML models or other structured information sources [4], [19]. This leads to the conclusion that the schema flexibility of the *vertical store* approach is not needed to efficiently capture most of the data in AWARE. In fact, it may even be detrimental, as the clearly structured data is broken up into generic $(subject - predicate - object)$ relations. If most reasoning rules or modules then require the original structured representation anyways, additional, unnecessary preprocessing work must be performed. If the schema flexibility of a vertical storage layout is needed for outputs of certain modules, it may still be applied selectively to those relations instead of a horizontal layout.

A major disadvantage of using a *type store* is the need to identify related attributes to create well-structured relations. To achieve this, various techniques can be used, which include analyzing access patterns of applications or data mining techniques [32]. Additionally, domain knowledge is beneficial when deciding on the scope of relations. In AWARE, the RDF knowledge graph is based on defined schemas and vocabulary anyways, which largely bypasses the need to identify hidden dependencies. Therefore, for AWARE, a *type store* approach is considered the most feasible solution.

In this work, the contextualization of data is limited to the temporal dimension for flights. The relation schema for flights includes a flight number as a unique identifier and dependent attributes that further capture flight and aircraft information. The attributes for flights can be separated into two variants. One variant is highly likely to change with each context change; those attributes will be denoted as a_{change}. The second variant, a_{fix}, could potentially change with the context, but it is rather unlikely. For example, the position of the aircraft will most likely change with each context, while the destination will almost never.

Consider a similar contextual representation to the one shown in Figure 20. A context change in a relation, with the schema $R_{flight}(a_{change\_1}, \ldots, a_{change\_n}, a_{fix\_1}, \ldots, a_{fix\_n})$, will include all rarely changing attributes $(a_{fix\_1}, \ldots, a_{fix\_n})$ in each tuple, even though they do not change their value. This may warrant a split of the relation R_{flight} into R_{flight_change} and R_{flight_fix} to reduce redundancy. As a drawback, the split would result in one additional join to reconstruct the original relation when needed and additional insert complexity for data, as inserts into R_{flight_fix} will violate its primary key most of the time. For the purposes of this work, the negative impact of splitting R_{flight} is considered to outweigh the positive impact of reducing redundancy. Figure 23 shows a section of the RDF graph

that serves as the initial situation to capture flight states in AWARE, as it is the proposed variant of the preceding AISA project and its mapping to a contextualized relational representation using a *type store* approach. The KG used in AISA uses named graphs as a contextualization method; this approach is equivalent to the described updated contextualization in Figure 20. The relation $flight$ has the $icao$ and the $timestep$ as primary key. The $timestep$ serves as the context attribute and is conceptually equivalent to the $valid\_from$ context used in previous sections and to the $requestTime$ context used in the example for the RDF graph capturing flight states in Figure 23.



relational representation

Flight relation				
icao	**timestep**	**origin country**	**lat**	**lon**
3ccc7f	1	Germany	47.5086	9.3028
471efb	1	Hungary	51.6735	8.0137
...	...	...	...	

Figure 23: Mapping of a contextualized KG using a named graph to a relational representation. The temporal context requestTime is mapped to the temporal context timestep of the flight relation.

8 Relational representation and implementation of rules

This chapter focuses on the handling of different types of relations and on considerations on which storage and mapping options are practical for the use case in this work. Section 8.1 covers a brief description of what is understood as a base relation in this work, followed by concrete base relations used for fact derivation by the implemented rules. Previously it was shown that rules may be mapped to *views*, *materialized views*, and *tables*. In Section 8.2, the practicality of using one data object over another to represent derived relations in the proposed *type store* layout is investigated. Specific implementations of the two investigated rules, in the relational system, are covered in Section 8.4. Finally, Section 8.5 describes other alternative approaches to representing relations that were not investigated in this work in detail but may still prove viable.

8.1 Base relations

In a database, base relations are considered tables, which store data without deriving it from other relations. Base relations in a relational setting are equivalent to EDB relations in Datalog. Both hold materialized data used for fact derivation by rules of the IDB in Datalog or derived relations in a database setting. Base relations are not necessarily static and may still be updated with data from external sources. Also, a base relation may be context dependent. Consider a relation with a schema $R_B(k_1,\dots,k_n,a_1,\dots,a_n)$. The attributes $(k_1,\dots,k_n)$ form the primary key, whereas attributes $(a_1,\dots,a_n)$ are not part of the primary key. R_B is considered a base relation if no attributes of R_B are derived from other attributes in R_B or any other relation. A set of attributes $(b_1,\dots,b_n)$ in R_B may still reference a set of attributes $(p_1,\dots,p_n)$ in another base relation R_P via a foreign key reference. However, $(b_1,\dots,b_n)$ must not be derived by performing derivation tasks based on the values of $(p_1,\dots,p_n)$, such as, for example, logical reasoning, number crunching, or string manipulation. This means that if $(b_1,\dots,b_n)$ references $(p_1,\dots,p_n)$, then $R_B(b_1,\dots,b_n) = R_P(p_1,\dots,p_n)$.

In the scope of this work two different base relations are required. The relation with the schema $R_{flight}(icao, timestep, a_1,\dots,a_n)$ represents tracked flights retrieved from the OpenSky Network. The attribute $timestep$ represents the temporal context of the relation in the form of an incrementally increasing integer value. Context for relations is further only represented as a single-level, one-dimensional context, but it could be expanded by additional levels and dimensions as shown in Section 7.3.2. R_{flight} has $(icao, timestep)$ as its primary key. The attributes $(a_1,\dots,a_n)$ are not context attributes, nor are they part of the primary key. However, they are all dependent on the context attribute $timestep$ and its value in the same tuple. From a conceptual perspective, the $timestep$ is similar to a $valid\_from$ context attribute, so that a tuple $< icao, timestep, a_1,\dots,a_n >$ is considered as a *current* representation of a flight if there is no tuple $< icao, timestep_2, a_1,\dots,a_n >$ in the relation R_{flight}, where $timestep_2 > timestep$. The *current* representation of a flight may be considered as stale if there is a tuple $< icao_{new}, timestep_{new}, a_{1_{new}},\dots,a_{n_{new}} >$ so that $timestep_{new} > timestep$. As flight data is captured from the OpenSky Network in a time interval for all flights simultaneously, if the maximum timestep of f_1 is greater than the maximum timestep of f_2 it indicates missing data for flight f_2. In the scope of this work, the business-logic implications of this special case of missing all data

Co-funded by the European Union

of a flight for a given timestep are not considered any further. If only partial data of a flight is missing for a given timestep, for example regarding its position, associated attributes are assigned NULL values.

The second base relation, $R_{airport}$, captures static data about airports. Airport data does not change over the course of an experiment and is context-independent in the scope of this work. Relation $R_{airport}$ follows the schema of $R_{airport}(airportid, a_1, \ldots, a_n)$, with $airportid$ as primary key and $(a_1, \ldots, a_n)$ as context-independent attributes.

8.2 Derived relations

In Datalog, derived relations of the IDB are not explicitly stored. This would make them conceptually equivalent to relational views [42]. However, as described in Section 6.4, relations derived by rules do not require a mapping to views. Alternatively, to meet demands of a use case, it may be better to map derived relations to materialized views or tables, instead of regular views. For example, for the investigated use case in AWARE, additional requirements concerning the availability and derivation performance must be met. In terms of availability, context-dependent, derived facts of 120 consecutive timesteps must always be available. For derivation performance, the derivation must be finished in under 1 second to accommodate a 1 Hertz refresh rate for real-time operation. To accommodate those requirements, materializing derived relations in materialized views or tables is considered. This section describes the scope in which each of the three variants may be practical.

8.3 Views for derived relations

Implementing Datalog rules as views is the most straightforward way. Available data is inserted into base tables as soon as it is available. The derivation starts when derived facts are queried by an actor or an application. This couples the derivation to a data request, which might not be feasible depending on the use case and the complexity of the query. For example, given a time-critical environment and a time-consuming calculation of derived facts, calculation only starts when the view is queried and the derived data is quasi already needed. Alternatively, it may have been possible to already calculate derived facts as soon as the base data is available and materialize the data in advance. Precalculation and materialization may be preferable, especially when dealing with recurring reasoning tasks.

Consider an ATCO tasked with monitoring the positions of aircraft and their short-term future positions, derived by the previously described *Rule 2*. The current and predicted positions of all tracked aircraft are displayed on the ATCO's screen. The application, used by the ATCO, queries the database for base data and derived facts in a 1-second interval. When using views, this triggers the derivation of facts and returns the requested data. Most time required to finish the execution of a query is likely ascribed to fact derivation. From a user perspective, the query takes a total time of $t_{query} = t_{fact\_derivation} + t_{select}$. When using a table-based approach instead, facts may already be derived as soon as the base tables have changed. From a user perspective, the query then takes a total time $t_{query} = t_{select}$. Figure 24 shows two flow charts, covering a view-based approach and a table-based approach. Flows are separated into groups a and b, where flows in a are decoupled from the data-requesting query and flows in b are not. The problem of coupling fact derivation to the query issued by an actor may be disregarded for rules with a low time consumption; however, it is reinforced when rules are chained. A rule r_n may require derived facts of an arbitrary number of preceding rules $r_1, \ldots, r_{n-1}$. Users querying the relation derived by r_n would perceive the time taken by the query as

Co-funded by the European Union

$\sum_{i=1}^{n} t_{fact\_derivation}(r_i) + t_{select}(r_i)$. For the table-based approach, the perceived querying time would not change for the user, from $t_{query} = t_{select}$, regardless of the number of preceding rules, under the assumption that fact derivation has already finished by the time the user queries the data.

Another drawback of using views arises when derived facts are queried by multiple actors. Each query to the view triggers fact derivation individually, even though base data has not changed since the last derivation. In this case, the time needed to perform recalculations repeatedly, will likely outweigh the time needed for materializing derived facts.

Data derived by the rules investigated in this work is dependent on a temporal context. It is necessary to always provide derived data for the past 120 timesteps, as future tasks and rules will depend on this derived data. The rules investigated require only the current temporal data, indicated by $max(timestep)$ in the relation $R_{flight}(icao, timestep, a_1, \ldots, a_n)$.

However, consider a use case where a user requires derived data of a rule r_1 lying in the past via a rule r_1^-. For example, instead of $max(timestep)$, an actor queries derived data of r_1 for timestep $max(timestep) - 10$. Ideally, the view would allow a parametrization at query time to set the required context dimension value. However, parameterized views are not available in PostgreSQL. If rules are represented as views, this requires the implementation of a specific view, representing r_1^-, that derives facts based on $max(timestep) - 10$. Alternatively, r_1 could be implemented to disregard its dependency on the relative temporal context value. It would derive facts for each available context dimension value of the temporal context dimension in the base relation R_{flight}. The view is referred to as *historical view*. The querying actor or application has to filter the returned relation for data concerning the required context dimension values. From a conceptual perspective, the resulting workload depends on the range of context dimension values of the temporal context. Considering the 120 timestep requirement for the use case in this work, this would result in approximately 120 times the workload when compared to the original proposed r_1, which renders the approach impractical from a purely conceptual perspective. If a materialization approach would have been used, the derived data would be available without defining a view for r_1^-, as the data would have been derived and materialized when $max(timestep) - 10$ was the *current* timestep.

However, when using a *historical view*, instead of deriving facts over the whole value range of context dimensions and then filtering the resulting relation, the optimizer of the DBMS may perform query rewriting. The, from a conceptual perspective, iteratively executed nested queries are flattened to a single query over the base relation, which prevents derivation of facts not part of the filter condition. As this approach is not necessary to achieve the performance goals in the use cases investigated in this work, associated in-depth performance benchmarking was not performed. However, a brief exploratory investigation showed a similar performance to the view-based approach used for benchmarking. The query-rewriting approach using a historical view comes with the advantage of more querying flexibility and easier maintenance of fact derivation logic, as the dependency on context dimension values is not included in the definition of the historical view. This approach separates the fact derivation logic from the filtering on context dimension values. Consider a change in the calculation method used for the calculation of distance. In the query-rewriting approach, an adaptation of the historical view may be sufficient as long as the resulting attributes remain identical, whereas the regular approach requires an adaptation of the definitions of all associated view variants that differ only in their restriction of context dimension values. However, a disadvantage is that the query-rewriting approach relies on the query optimizer. It cannot be guaranteed that query rewriting is performed for an arbitrarily deep nesting of rules.

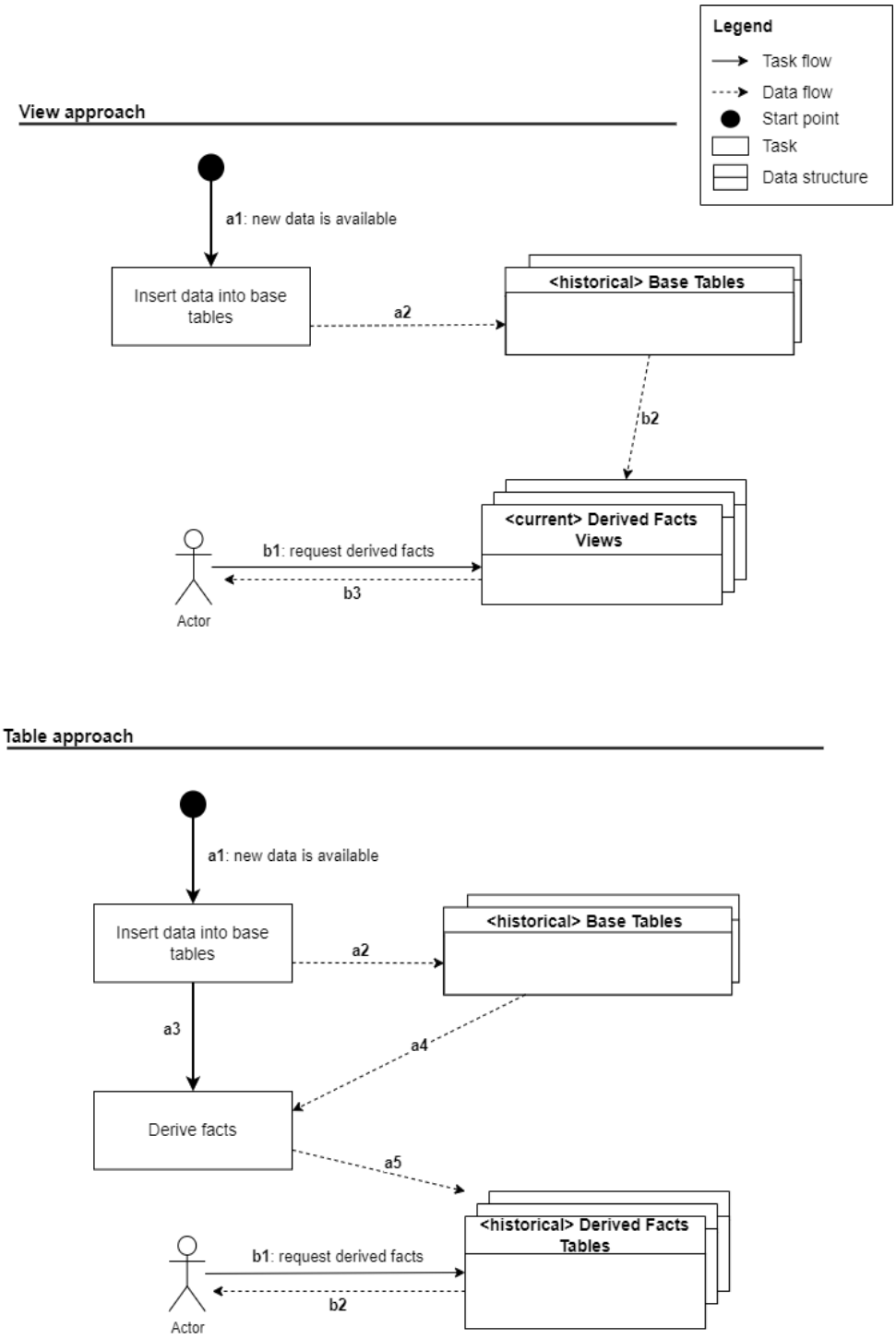


Figure 24: Comparison of data flow and tasks between an approach using only tables and an approach using views. Flows within a group occur according to their numeration. Flows of Group *a* are performed independently of a querying actor, whereas flows of Group *b* start with an actor requesting the derived data.

A potential alternative would be to use historical tables to materialize facts after their derivation, instead of performing recalculation of past facts when needed. History tables are then provided for use cases where derived data, based on past context values, is required. However, if persistence or materialization of data is needed anyways, using alternatives, as, for example, materialized views or temporal tables, should be considered.

8.3.1 Materialized views for derived relations

To counter the drawbacks of lacking materialization when using views, materialized views may be used. Providing temporal context to materialized views roughly incorporates two major steps. The first step corresponds to refreshing the view by recalculating derived facts and materializing them. The second step corresponds to providing the data to a querying actor, even during fact derivation. Materialization of derived facts prevents a recalculation at querying time, similar to the use of tables instead of views, shown in Figure 24. Nevertheless, recalculation performance is likely to decrease when compared to *views* due to the additional materialization of derived facts. Recalculation performance becomes even worse when concurrent multi-actor activity must be accounted for.

PostgreSQL's materialized views provide a *concurrent* mode for refreshing materialized views. This mode keeps the materialized view responsive for concurrent selects and its data available for actors who access the data during the refreshing process. Otherwise, actors who try to access the old materialized data during the refreshing process must wait for the materialization of the new data to finish. Depending on the use case, preventing access to the data of the materialized view during refreshing might be infeasible. This is especially problematic when a certain response time of the system is required, and refreshes are performed with high frequency. For example, the use case investigated in this work requires a 1 Hertz refresh rate of derived facts, and derivation must be finished in under 1 second. Therefore, a concurrent refresh of materialized views is required. Otherwise, unfavorable timing of queries, executed by actors, may finish with unexpected delay as the queried data is currently not available. This would defeat the purpose of materializing data for fast response time.

Opposed to views, materialized views offer a solution to the problem of additional recalculation of derived facts in case of multi-user access. However, they do not solve the problem of accessing past temporal derived facts for further use. A materialized view is created in a similar way as a traditional view with the addition of the *materialized* keyword. Access to previous temporal states of the materialized views would again require an implementation of a specific materialized view that derives facts based on the specific past temporal context value. A solution is again to use a history table to archive temporal states of the materialized view before refreshing the data, similar to what was proposed for views. The difference of archiving temporal view states opposed to using tables is shown in Figure 25.

When using historical views, described previously in Section 8.3, a reasonable alternative may be to materialize the historical view. Materializing a historical view as r_m has the advantage, that its derived relation R_m can be treated as a base relation for other depending rules. Query rewriting is then used to flatten nested views to the derived relation R_m and not to a true base relation. This approach reduces the number of required rule executions when a rule that depends on r_m is executed, as the results of all derivation rules preceding r_m are already available in the materialized relation R_m. To maintain an overall similar performance to a view approach without materialization, it is assumed that

the DBMS supports incremental view maintenance (IVM). When a materialized view is refreshed via IVM, only updates to the materialized view are performed that are needed to reflect changes in data to the last refresh. Without IVM, R_m is completely derived from scratch every time its refresh is triggered. If base data changes frequently and rules that depend on r_m require latest data, the refresh of R_m will likely become impractical. When using virtual views instead of materialized views, the same issue of needing to derive all data in the historical view is mitigated by query rewriting. However, when the historical view is materialized, a complete derivation of R_m is necessary, even though depending rules may not require a majority of the data.

Figure 25 shows a comparison of data flows between an approach using only temporal tables and the approach of using materialized views. The materialized view approach, if the materialized views are implemented in the proposed way, introduces additional complexity to the overall process without generating benefit. However, consider again an approach using historical materialized views. Another virtual view or materialized view can now be defined on top of the historical materialized view, which is based on a specific context, for example, $max(timestep)$. When new data is inserted into the base relation and the historical materialized view is refreshed to derive facts, as the definition of the historical materialized view does not include an aggregation, IVM may be employed. By using a historical materialized view with IVM, it is not anymore necessary to archive derived facts before refreshing the materialized view, as shown in Figure 25. This reduces the complexity of the task flow and data flow and may improve performance. In PostgreSQL, IVM is not available out of the box and would require an additional extension. The extension is relatively new and is also subject to some restrictions, for example on view definitions, but it is actively developed [49]. Whether the extensions maturity would suffice for a use in a productive setting of an enterprise, or for the use in exploratory research is difficult to determine and out of scope in this work. Therefore, the investigation of the performance and practicality of IVM in PostgreSQL remains for future research.

8.3.2 Tables for derived relations

The final way of handling derived relations is the use of tables. How Datalog rules are mapped to relational tables was previously shown in Section 6.4. In this case, Datalog rules that derive data are not mapped to views or materialized views, but derived facts are stored explicitly in a relational table.

Consider a rule r_{tr} that derives the trajectory of all flights provided by a base relation with the schema $R_{flight}(id, C_t, a_1, \ldots, a_n)$. For simplicity, assume a single-level, one-dimensional context in the time dimension, C_t, for all relations. However, as shown in Section 7.3.2, the scope of the provided context can be expanded to multiple levels and dimensions. Derived facts are inserted into a derived relation with the schema $R_{trajectory}(id, C_t, d_1, \ldots, d_n)$. To detect dangerous situations early and prevent collisions of aircraft, r_{tr} is executed as soon as an external *ADS-B* module provides new aircraft positions to the flight base relation. The value of C_t for a given tuple in R_{flight} is a timestamp indicating the insert time. The value of C_t for a given tuple in $R_{trajectory}$ is the value of C_t of the base data that is used for its derivation. Assume a necessity to provide previously derived data, for example, to provide replicability for controlling measures or to provide data for other dependent rules. Trajectory derivation is a recurring task, executed each time new base data becomes available. Derived facts are materialized in a relational table. Therefore, it is not necessary to re-derive facts over all available context dimension values of the base relation. It is sufficient to derive facts associated with only the last introduced context dimension values. For r_{tr}, this means fact derivation is only performed for the last introduced timestamp in R_{flight}.

Using a materialized view for fact derivation requires the use of historic tables to achieve similar capabilities with respect to the proposd requirements. Figure 25 shows a comparison of data flows between an approach using only derived fact tables and the approach of using materialized views. Comparing the two implemented approaches of using derived fact tables and using materialized views, the derived fact table approach is simpler from a scheduling, maintenance, and development perspective.

Similar to materialized views, triggering fact derivation is decoupled from querying derived data by an actor or application. Various ways of starting fact derivation can be realized, as, for example, using *AFTER INSERT* database triggers on corresponding base relations, scheduled recurring cron jobs, or by notifying client applications.

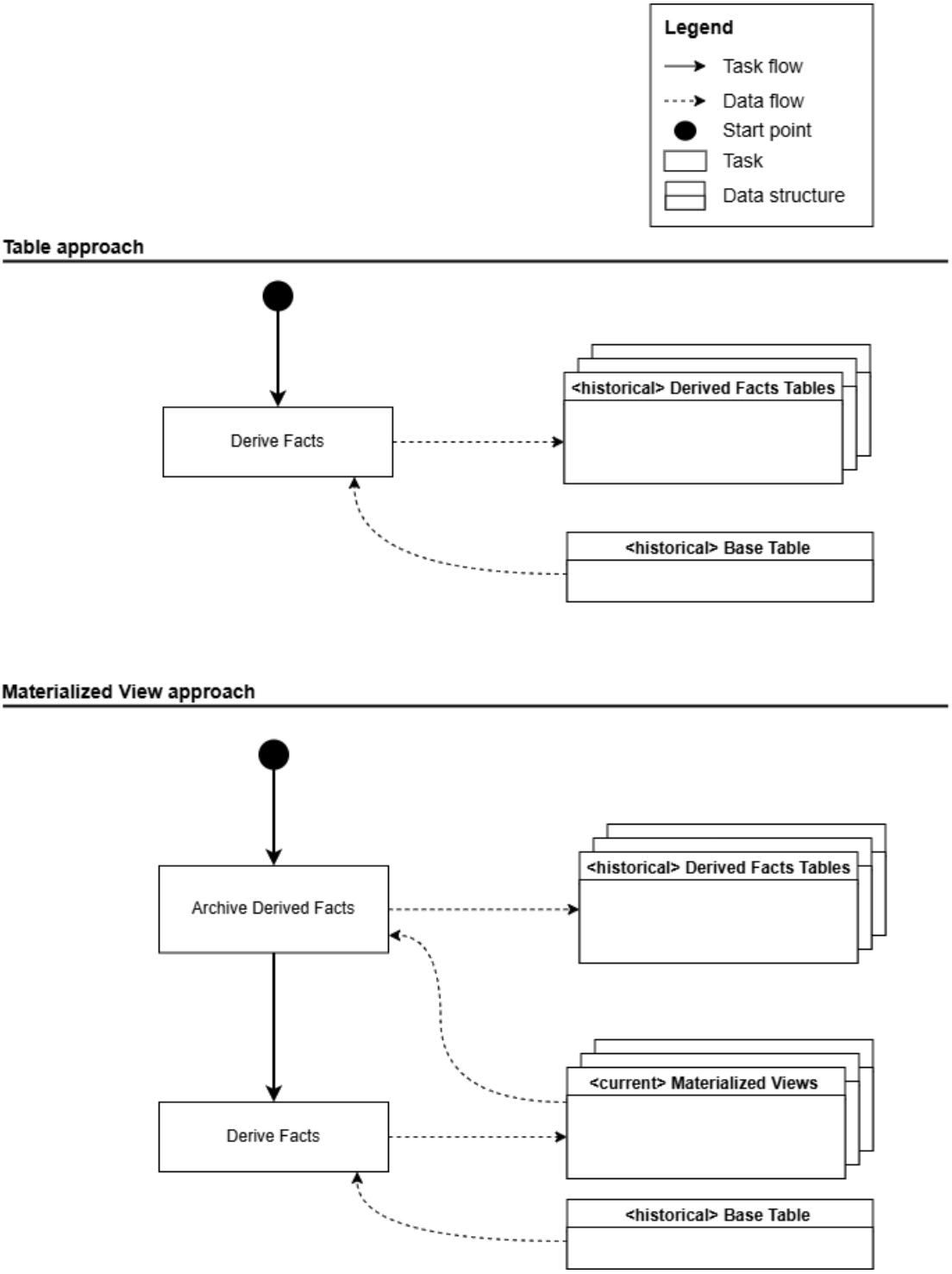


Figure 25: Comparison of data flow and tasks between an approach using only tables and an approach using materialized views.

8.4 Rule implementations

This section covers different approaches to deriving facts of the two implemented rules described in Section 4.2. In the preceding AISA project, calculations that are needed for fact derivation were implemented in Java. The reason for using Java instead of pure Datalog or Prolog was the complexity and nature of the rules. Given the two described rules in Section 4.2 and the reasoning tasks previously implemented for AISA, it is found that most of the reasoning complexity stems from comparisons and calculations instead of logical reasoning. Rules that mainly gain their complexity from calculations are in this work referred to as *number crunching* rules. During discussions with the project partner, it was not possible to specify the exact share of complex logical reasoning tasks in the AWARE project. However, as the implementation of a relational architecture would at least partly substitute the existing KG system, and especially for *number crunching* rules, an integration of Java into the relational architecture is considered. Using Java for calculations regarding the implemented rules would decrease the costs of adaptation to the new system, as program code for calculations of derived facts could be reused partially.

For an integration of Java into PostgreSQL, the *PL/Java* extension was added. *PL/Java* is a free open source extension that allows for stored procedures, triggers, and functions written in Java to be directly executed in the PostgreSQL backend. This is made possible via an efficient and integrated version of the standard Java JDBC API. This approach eliminates the need for an additional, separately maintained Java application.

8.4.1 Rule implementations – Rule 1

In this section the concrete implementations of Rule 1 are described. The section is separated into two parts. The first part covers performing the number crunching via the previously described Java extension, whereas the second part focuses on a purely SQL-based approach for number crunching.

To use the Java variant for *Rule 1*, a Java class with a *simplifiedDistance* method was implemented. The implementation is shown in Listing 19. The function takes latitude, longitude, and altitude values of two points and calculates the equirectangular distance as the horizontal distance. Then the Euclidean distance calculation is used to include altitude differences. It is important to note that the function in Listing 19 returns only a single value for the overall distance between two provided points. This behavior is not similar to the one proposed by Steiner [41] or implemented in the native approach in Section 5.3.1. In both cases the distance is captured as three separate triples: the horizontal, vertical, and overall distance [1], [41]. To provide comparability to the works of Steiner [41] and to the native approach, an equivalent variant was also implemented, which is shown in Listing 20. As calculations are mainly similar on a surface level, a similar performance was expected for both variants. This encourages the use of the combined calculation in Listing 19 if only the overall distance is needed and using the calculation from Listing 20 if the use case requires each or only certain parts. Using the combined calculation also requires a schema change of the derived relation. However, using the separated variant in Listing 20 requires three Java function calls instead of one. This results in two additional context switches per tuple. The performance difference of the two approaches is shown via experiments in Section 10.2.

```
public class SimplifiedDistance{
    @Function(onNullInput=RETURNS_NULL, effects=IMMUTABLE)
    public static double simplifiedDistance(double lat1,
        double lat2, double lon1, double lon2, double al1,
        double al2) {

        final int R = 6371; // Radius of the earth
        double x =
            (lon2 - lon1) * Math.cos((lat1 + lat2)/2);
        double y =
            lat2 - lat1;
        double horizontalDistance =
            Math.sqrt(Math.pow(x,2) + Math.pow(y,2)) * R;
        double verticalDistance = al2 - al1;

        return Math.sqrt(Math.pow(horizontalDistance,2) +
            Math.pow(verticalDistance,2));
    }
}
```

Listing 19: Simplified distance calculation

```
@Function(onNullInput=RETURNS_NULL, effects=IMMUTABLE)
public static double hdistance(double lat1, double lat2,
    double lon1, double lon2) {

    final int R = 6371; // Radius of the earth
    double x =
        (lon2 - lon1) * Math.cos((lat1 + lat2)/2);
    double y = lat2 - lat1;
    double horizontalDistance =
        Math.sqrt(Math.pow(x,2) + Math.pow(y,2)) * R;

    return horizontalDistance;
}
@Function(onNullInput=RETURNS_NULL, effects=IMMUTABLE)
public static double vdistance(double al1, double al2) {
    return Math.abs(al2 - al1);
}
@Function(onNullInput=RETURNS_NULL, effects=IMMUTABLE)
public static double odistance(double hd, double vd) {
    return Math.sqrt(Math.pow(hd,2) + Math.pow(vd,2));
}
```

Listing 20: Simplified distance calculation – separated

The Java functions previously described may alternatively be implemented in pure SQL without the need for Java. Considering maintenance efforts and complexity of design, using only SQL instead of including Java via the *PL/Java* extension seems beneficial. Additionally, regarding performance, starting the Java virtual machine and repeatedly switching context to the user-defined function to perform small calculation tasks has a negative impact on performance. In this work, the pure SQL variant is tested via inline calculations. Another possible approach would be to use user-defined functions, but instead of Java, using *PL/pgSQL* or other available procedural languages. Performing inline calculations is preferable, as the complexity of calculations is low and calculations are not reused in other rules.

Similar to the Java-based approach, the purely SQL-based variant is also implemented once as a combined calculation in one resulting distance attribute and also once in a separated calculation yielding three different distance values. Listing 21 shows the calculation of the overall distance as one derived attribute, whereas Listing 22 shows the same calculation with the overall distance separated into horizontal, vertical, and overall distance. Separating the calculation of the derived distance attributes causes a higher complexity of SQL code. Calculation of the overall distance is performed via the same code in both approaches. The horizontal and vertical distances are captured in the attributes *hdistance* and *vdistance*. To calculate the overall distance, some code duplication is necessary, as *hdistance* and *vdistance* cannot be reused in the same *SELECT* clause of the SQL statement. However, due to optimization by PostgreSQL, the performance impact is negligible.

```
(sqrt(
    power((sqrt(
            power(((p.longitude - f.longitude) *
                cos((f.latitude + p.latitude)/2)), 2)
            +
            power((p.latitude - f.latitude),2)
            ) * 6371)
        ,2)
    + power((p.elevation_ft - f.geo_altitude),2)
    )
) as distance
```

Listing 21: Simplified distance calculation SQL

```
(sqrt(
    power(((p.longitude - f.longitude) *
        cos((f.latitude + p.latitude)/2)), 2)
    +
    power((p.latitude - f.latitude),2)
    ) * 6371) as hdistance,
(p.elevation_ft - f.geo_altitude) as vdistance,
(sqrt(
    power((sqrt(
            power(((p.longitude - f.longitude) *
                cos((f.latitude + p.latitude)/2)), 2)
            +
            power((p.latitude - f.latitude),2)
            ) * 6371)
        ,2)
    + power((p.elevation_ft - f.geo_altitude),2)
    )
) as odistance
```

Listing 22: Simplified distance calculation SQL – separated

1.1.2 Rule implementations – Rule 2

In this section the concrete implementation of Rule 2 is described, which follows a similar approach to Rule 1. The section is again separated into two parts. The first part covers performing the number crunching via the previously described Java extension, whereas the second part focuses on a purely SQL-based approach for number crunching.

Rule 2 uses the current position data, the direction the plane is facing, and the current speed the plane is flying to calculate the future position of the plane. It is important to note that *vertical rate* and *velocity* describe the change in altitude and the horizontal speed, respectively. Given this current position and speed, the rule derives the position of the airplane 10 seconds in the future via the calculation of three values. The values *predLat*, *predLon*, and *predAlt* represent the predicted latitude, longitude, and altitude values. The predicted position can in this case only be described by those three attributes and no equivalent single attribute, as was the case in *Rule 1* described in Section 8.4.1, which keeps this rule inherently comparable to equivalent implementations of Steiner [41] and to the native approach.

To use Java in the derivation of the position prediction attributes, three Java methods, each in one separate Java class, were implemented. The Java methods take the needed values for location, speed, and a constant representing the seconds in the future for which the prediction is calculated. The calculation of predicted values is based on the work of Steiner [41] and is shown in Listing 24.

Similar to the implementation of *Rule 1* in Java, which is described in section 8.4.1, starting the Java virtual machine and switching contexts to perform small calculation tasks has a negative impact on performance. This problem is especially prevalent for *Rule 2* as it is not possible to represent and

calculate the three derived attributes alternatively as one single attribute. This causes context switches for each of the three attributes per tuple without an option to reduce overhead as in *Rule 1*, where it would be possible in use cases where only the overall distance is needed.

Similar to *Rule 1*, the Java functions for *Rule 2* can again be alternatively implemented via pure SQL instead of Java. Choosing an implementation in only SQL instead of using the *PL/Java* extension again reduces the overall potential maintenance efforts and complexity. *Rule 2* is implemented as inline calculations, foregoing the implementation of user-defined functions, as the calculation complexity is low and calculation logic is not reused in other rules. Listing 23 shows the derivation of *predLat*, *predLon*, and *predAlt* via inline SQL-based calculations in a *SELECT* clause of a statement.

```
latitude + degrees((velocity * 10) * cos(heading) /
    6371) as predLat,
longitude + degrees((velocity * 10) * sin(heading) /
    (6371 * cos(radians(latitude)))) as predLon,
(altitude + vertical_rate * 10) as predAlt
```

Listing 23: Calculation of position prediction in SQL

```
public class PredictLat{
    @Function(onNullInput=RETURNS_NULL, effects=IMMUTABLE)
    public static double predictLat(double lat,
        double heading, double velocity, int timeSec) {

        final int R = 6371; // Radius of the earth

        double distance = velocity * timeSec;
        double deltaLatitude = distance *
            Math.cos(heading) / R;
        double predLatitude = lat +
            Math.toDegrees(deltaLatitude);

        return predLatitude;
    }
}

public class PredictLon{
    @Function(onNullInput=RETURNS_NULL, effects=IMMUTABLE)
    public static double predictLon(double lat,
        double lon, double heading, double velocity,
        int timeSec) {

        final int R = 6371; // Radius of the earth

        double distance = velocity * timeSec;
        double deltaLongitude = distance *
            Math.sin(heading) /
            (R * Math.cos(Math.toRadians(lat)));
        double predLongitude = lon +
            Math.toDegrees(deltaLongitude);

        return predLongitude;
    }
}
public class PredictAlt{
    @Function(onNullInput=RETURNS_NULL, effects=IMMUTABLE)
    public static double predictAlt(double alt,
        double verticalRate, int timeSec) {

        double deltaAltitude = verticalRate * timeSec;
        return alt + deltaAltitude;
    }
}
```

Listing 24: Java functions for position prediction

8.5 Alternative implementation approaches for derived relations

The two rules that are investigated in this work are based on a single context dimension $time$ with one level $timestep$. Data is repeatedly inserted into the base relation with incrementally increasing values for a $timestep$ attribute. The rules as implemented derive facts based on the newest available $timestep$ value. If a materialization approach is used for derived facts, via materialized views or tables, derived facts of $timesteps < max(timestep)$ are materialized in a historical table. This approach shows high performance for the use case investigated in this work. However, using an aggregate function to determine the latest value of the $timestep$ attribute may become impractical, from a performance perspective, when additional constraints on other context dimensions are required, or when rules depend on other derivation rules in a multi-layered nested structure. In this section, two alternative implementation considerations are described briefly. The first consideration uses a *context table* to substitute the aggregate function by a join. The second consideration concerns the use of nested views to provide inherent scheduling of iteratively executed rules, without needing to develop, or maintain, custom scheduling flows. The performance of the alternative considerations are not investigated in detail, but brief exploratory testing suggests a similar performance to the investigated approach that uses the aggregate function.

8.5.1 Context table approach

Consider a relation R_c with a schema $(c_1, \ldots, c_n)$, that represents specific contexts C as tuples. Each attribute in $(c_1, \ldots, c_n)$ specifies an available context dimension. A context dependent rule $r1$ may substitute a strict binding to a specific context in its rule definition, by for example an aggregate function, with a more dynamic binding to contexts represented as tuples in R_C, by using a join on matching context dimension values. A single context table may serve multiple context-dependent rules. For *Rule 1* and *Rule 2* in this work, the context table would only hold a single tuple at all times that includes only the greatest $timestep$ as a single context dimension value.

Consider an actor that has access to an implemented rule r_2 and an associated context table R_{C2}. The rule r_2 is implemented using a view and derives facts based on tuples in a base relation with the schema $R_B(id, c_1, \ldots, c_n, a_1, \ldots, a_n)$, by joining R_{C2} and R_B on $(c_1, \ldots, c_n)$. If the actor requires derived facts based on other context dimension values, the tuples in R_{C2} are used to reflect those requirements. After changing the tuples in R_{C2} the actor may use the same rule r_2 to derive the facts based on the new context dimension values. This approach poses a solution to the previously described issue of needing to define separate views for individual contexts. The approach allows for a parametrization of views according to the tuples in R_{C2}, however changing the context for r_2 then requires additional intervention before executing the rule, for example by an actor, an external application, or a trigger. This increases the complexity of maintenance and scheduling of rules, when compared to a single, static rule.

8.5.2 Nested views for iteratively executed rules

Consider, rules that are dependent on the results of other preceding rules so that a rule r_n depends on rules $r_1, \ldots, r_{n-1}$. As described in Section 8.3 when using views, from a user perspective the time consumption of querying r_n includes the sum of consumed time for all fact derivations associated with the preceding rules $r_1, \ldots, r_{n-1}$. In comparison, when using an approach that materializes derived facts of $r_1, \ldots, r_{n-1}$ and r_n the perceived time consumption is reduced to the selection of required facts

which were derived by r_n. Even though the query optimizer of the DBMS may use query rewriting to flatten nested views to a single query on the base table, the underlying derivation logic must be performed for each rule. This will occur a worse performance than selecting already materialized facts.

However, using a view approach instead of a table approach simplifies scheduling concerns of rule executions. By using nested views, rules that must be executed in advance, are included in the view definition and may be nested in arbitrary depth. If a table approach is used instead, as described in Section 8.3.2, correct scheduling cannot be automatically achieved by the definition of the rules. Therefore, custom scheduling efforts must be made, for example by using an external scheduling application, database triggers, or a recurring *cron-job*. Developing, maintaining, and expanding custom scheduling flows may be reasonable for small use cases, however with increasing scope, this approach may become impractical. From a conceptual perspective, nested materialized views may be used instead of nested views. However, from a scheduling perspective, a refresh of materialized views must be triggered separately, which may again be impractical for large-scale use cases.

9 Handling dynamic data in the relational system

During operation of the system, new flight data is continuously inserted into the database, and new data is derived via implemented rules. As the system requires responsiveness in real time and complete availability is assumed, it is necessary to delete old, obsolete data during operation. This chapter describes different implemented and tested deletion strategies.

9.1 No deletion

As a first strategy, it is considered to completely ignore the potential negative performance impact of a huge data set. If obsolete data is never deleted, the data set is steadily growing and will eventually reach a size where the performance of operational tasks drops below the minimum requirement. For a system operating in a productive enterprise setting, choosing this strategy is not feasible even though relational DBMSs are well equipped to handle large amounts of data. However, for the purpose of performing simulations and exploratory studies in the setting of this project, it may still be feasible and is therefore still investigated.

9.2 Deletion per insert via triggers

One approach to deleting old, obsolete data is to delete obsolete data as soon as new data is inserted into the knowledge graph. This deletion strategy closely aligns with the use case of this project, as the 120 newest timesteps have to be available at all times to cover all potential rules. To perform deletion on this condition, a monitoring of the data set is required. Monitoring and deletion of obsolete tuples could be achieved by a client program. This client would ideally monitor all relations of the database to prevent frequent reopening of many different connections to the database. Even though performing data handling via a client may be feasible for a small use case, with a growing data schema in an enterprise setting, the costs of maintaining the code base of one huge or a multitude of small clients increase.

As an alternative, conditional data handling as described may also be achieved by utilizing DBMS capabilities. By implementing after-insert triggers on the statement level for historical tables in the DBMS, a deletion of old data can be performed as soon as new data is available. Depending on use case-specific constraints, insertion of new data in the base table may not only trigger a data-handling routine in the base table but also in other dependent historical tables that contain derived data. However, this may lead to failing to maintain the minimum number of required timesteps in the historic derived data tables if the data-handling routine deletes data before the newest data is derived and materialized. Conversely, triggering data handling for multiple tables after finished data derivation of the last rule in a chain of dependent rules was completed may be feasible, again depending on use-case-specific constraints.

The most conservative approach is to provide one individual data-handling trigger per historical table. This approach prevents accidentally undercutting the minimum required number of available timesteps but also comes with the highest overhead, both in terms of code maintenance and trigger

invocations. The impact on performance when using this deletion strategy is investigated in the presented experiments.

9.3 Deletion in bulk via triggers

If the frequency of new data inserts is high, triggering the deletion of old tuples on each *INSERT* statement results in a high strain on the system. For example, given a separate trigger for each historic table, it would result in a relatively small deletion volume per trigger and would also incur a high transaction overhead. To reduce the overhead, reducing the frequency of triggering deletion procedures may be a viable option. Decreasing the frequency in turn increases the deletion volume. However, the negative performance impact of a higher deletion volume may be outweighed by the performance gain of reducing overhead and performing fewer context switches.

Therefore, the idea of triggering a deletion procedure not after each insert statement but only after available data surpasses a configured threshold was investigated. For the purpose of the experiments, a threshold of 20 timesteps was assumed. Ideally, checking whether the threshold is reached should be performed before performing the context switch to a stored, user-defined deletion procedure via a trigger. However, at the point of writing, PostgreSQL does not allow for such preemptive checking. This limitation leads to an implementation where an insert triggers the invocation of a deletion procedure that firstly checks whether the threshold is reached. If the threshold is not yet reached, the procedure terminates. Even though an early termination of the procedure saves resources that would otherwise be needed for deletion, the context switch and procedure calls are still performed, meaning the transaction overhead still occurs in the same way as described in Section 9.2.

9.4 Deletion in a time interval via a cron-job

The final explored deletion strategy is a deletion in a time interval via a cron-job. PostgreSQL supports an extension called *pg-cron*, which follows a similar idea as the *Cron-Daemon* on UNIX-like systems. *Cron* on Linux is a system scheduler that can be used to run a series of commands or scripts on a predetermined schedule [50]. Oftentimes such tasks are performed on systems that do not allow for downtime [50]. Commonly tasks performed via a cron-job include but are not limited to system maintenance, backups, and monitoring [50].

The PostgreSQL extension *pg_cron* allows the execution of commands directly from the database, which is preferable to cron-jobs implemented via the operating system based *cron*. It is preferable as data handling logic is then confined to a single source, the DBMS, which reduces future maintenance work. Additionally, it also reduces potential compatibility issues for future versions of the operating system. Also, by running maintenance tasks directly in the database, communication and authentication overhead is reduced.

For this architecture, when using the *cron*-based deletion strategy, a major advantage comes with the decoupling of data maintenance and continuous operation. Insert-trigger strategies impact performance negatively due to immediately triggering multiple data-handling procedures after inserts. Especially with an increasing size of the data schema and number of implemented rules, triggering the procedures causes multiple database connections to compete for resources simultaneously. Additionally, due to rather simple data handling procedures, the relative resource consumption of overhead tasks, like opening and committing transactions, is high in comparison to resources

consumed by actual procedure logic. However, when using a *cron*-job, which covers data handling for all, or at least a majority of the tables, the overhead is reduced to a single context switch, which saves resources for the data handling logic or rules for reasoning.

10 Experiments with the relational system

This chapter is separated into three parts. Section 10.1 describes the setup of the experiments and how performance is captured. In Sections 10.2 to 10.4 concrete experiment configurations and their results are highlighted. Section 10.2 compares different approaches in terms of using views, materialized views, and tables for derived relations and the different rule implementations. Section 0 focuses on different data handling variants. In Section 10.4, the impact of other aspects of configuring the PostgreSQL instance is investigated, for example, using a RAM-disk or indexation.

10.1 Setup

The experiment setup follows the general outline described in Section 4.4. Data is captured from the OpenSky-Network as RDF graphs and transformed to a *CSV* format. Airport data is considered static data and comprised of one single *CSV* file due to the lack of a temporal context. Flight data, on the other hand, is contextualized in the time dimension. Therefore, the data-capturing procedure yields a collection of RDF graphs, each capturing data of all flights at a corresponding point in time. Each RDF graph is separated into an individual file without additional temporal context information in the triples. However, the files are sequentially numbered via a timestep suffix. When transforming the provided RDF graphs into a *CSV* format, the timestep suffix is mapped to a newly introduced timestep attribute. This approach results in a data structure that enables the capturing of a temporal context based on the approach of *context-dependent relations* that was previously described in Section 7.3.2. The additional separation of temporal data through individual files is still maintained. This data structure enables a straightforward and repeatable simulation of continuously arriving new data by iterating over available *CSV* files in the file system. Using this file-based temporal structure does not have any impact on performance results.

Generated *CSV* files are additionally separated into two batches. The first batch is comprised of 120 individual files, which serve as the necessary data to simulate a warm start of the system under test. The second batch is then used for testing the performance. Batches are ordered according to their temporal context, which results in similar data over all experiments and configurations. It is necessary to reuse the same data over all runs of a single experiment to maintain comparability of results.

Generally, experiments are performed by Bash scripts, which are executed on the same machine that operates the PostgreSQL database. The performance impact of running the lightweight Bash scripts and their impact on time measurements is negligible. During resource-intensive data derivation, the script responsible for triggering the execution and tracking the time waits anyway. Another possible approach would be to run the scripts performing the experiments on an external machine. However, using this approach comes with the major downside of introduced network traffic, which is hard, or outright impossible, to exclude during time tracking. Additionally, incorporating time lost due to network traffic into performance testing reduces the reproducibility and validity of the performance evaluation, as network bandwidth and connectivity may strongly differ between hardware setups.

Due to the time intensity of running experiments and the high number of possible different configurations, the testing procedure is separated into three levels, each with associated Bash scripts. Figure 26 shows a visualization of all levels, the associated scripts, and the tasks performed. The *Run experiment collective*-script is used as a single point for starting an experiment with different

parameters. By setting parameter combinations inside the script, consecutive test runs are easier to adapt. The parameters *Data volume* and *Number of rules* concern the experiment workload. *Data volume* sets the number of airports or flights, whereas *Number of rules* sets the number of times the rule under test is duplicated. Rules are duplicated to test scenarios where multiple monitoring tasks are implemented in the system. The tasks that will be implemented are currently unknown. Therefore, experiments are performed with multiple independent duplications of the same rules. To prevent outliers from overly affecting the outcome of performance measures, a single experiment is repeated multiple times. The number of repetitions is described by the parameter *number of runs*. The default value for the experiments is ten. Outliers may, for example, be caused by background processes performed by the operating system and cannot be prevented entirely. Finally, different variants of representing derived relations may be investigated, leading to different calls of subscripts.

The underlying *run_experiment_generic* Bash scripts, which are called by the described *Run experiment collective* script, perform the workload according to the specified parameters and capture the consumed time as the performance measure. The parameters *Data volume*, *Number of rules*, and *Current run* are passed to the script at runtime, altering the workload or specific written logs. The parameter *derived relation representation* requires a different approach to the workflow for each tested variant, which led to the decision of separation into different scripts. Considering the generic structure of the scripts concerning most of the parameters, the scripts follow a naming pattern of *run_experiment01/02_generic* with an added suffix highlighting the *representation of derived relations* under test.

Concrete settings for *Data volume* and *Number of rules* directly influence the workload of the experiment. Choosing sets of configurations for both of those parameters results in a combination describing the workload for a specific experiment run. *Data volume* is described as sets. For *Rule 1*, sets describe the number of airports used for the cross join between flights and airports. For *Rule 2*, sets describe the number of flights. As *Rule 1* requires a cross join and *Rule 2* basically covers only number-crunching tasks based on one table, *Rule 2* performs generally faster. Therefore, the data volume is increased above the available flights by replicating the available data per timestep, even though resulting data sets can be considered as unrealistically high data volumes. The data replicating process was performed before conducting the experiment to prevent a potential influence on performance. Table 11 shows a breakdown of the *Data volume* for both rules. For *Rule 2* the *Data volume* in the Table 11 is expressed as a scaling factor, resulting in $\#flights_{rule02} = \#flights \times scaling\_factor$

Data volume

set name	airports for *Rule 1*	flights scaling factor for *Rule 2*
Set01	100	15
Set02	50	10
Set03	10	1
Set04	1	-

Table 11: Configurations for data volume across experiments

Number of rules covers six different configurations from 1 to 20 duplicated rules. For consistency the resulting parameter combinations of *Data volume* and *Number of rules* follow a naming schema of *C{Set number}_{Number of rules}*.

When a generic script is called, first a warm start scenario is established. Therefore, flight data corresponding to 120 timesteps is inserted. In a productive scenario this flight data would have also triggered the derivation of facts according to the specified rules. Therefore, derived data for 120 timesteps must be inserted too. As the same data set is used for each experiment and rules within an experiment are duplicated according to the *Number of rules* parameter, the derived data is calculated beforehand and reused over multiple different runs. The prepared data is inserted in bulk using the *COPY* command provided by PostgreSQL.

After a warm start scenario is established, flight data is repeatedly inserted, and facts are derived. In a productive employment of the system, different variants of triggering a start of fact derivation are conceivable. For example, using database triggers to start fact derivation after an insert occurs, using a timely recurring job that uses the latest available data, triggering fact derivation from an external scheduling application, or by a querying actor. For measuring the performance in this simulated system, triggering fact derivation is performed directly inside the script, closely aligning with the variant of a custom scheduling application, or with the variant of a querying actor.

Fact derivation is performed via using the PostgreSQL-specific command-line interface *pgsql*. For tracking the consumed time, the Bash command *time* is used. The command measures and returns the time that passes until a following command or code block finishes execution. An alternative for time tracking would have been to use PostgreSQL's internal time-tracking mechanisms. However, when performing time measures inside PostgreSQL directly, it does not represent the real time a user or external application would need to wait until requested data is available. Generally, in the performed experiments, the insertion time of the flight data is not included in the time measurement. The reason for this decision is that the time needed for inserting the rather low number of flights contained in each timestep is basically negligible, and most of the consumed time would be ascribed to transaction overhead. This transaction overhead is strongly dependent on the form in which the data is available. For example, time consumption differs highly when the data is inserted via one *INSERT* statement per tuple as opposed to a bulk insert spanning the whole data set. Ideally, the data would be available as a file in the file system of the server to access and insert the data via a *copy* command. Additionally, in a productive setting, data might be collected from different sources or require mapping from RDF into a relational form, which would both negatively affect performance. At the point of investigation, such specifics concerning data format and sources cannot be answered, which leads to the decision to generally exclude data insertion time in the experiments. However, to maintain comparability with the native system described in Section 5.5, configurations that were identified as the best-performing ones were additionally tracked with included data insertion time. This approach enables a fair comparison between the two technologies while preventing a dilution of general performance results due to assumptions of potentially unrealistic insertion strategies.

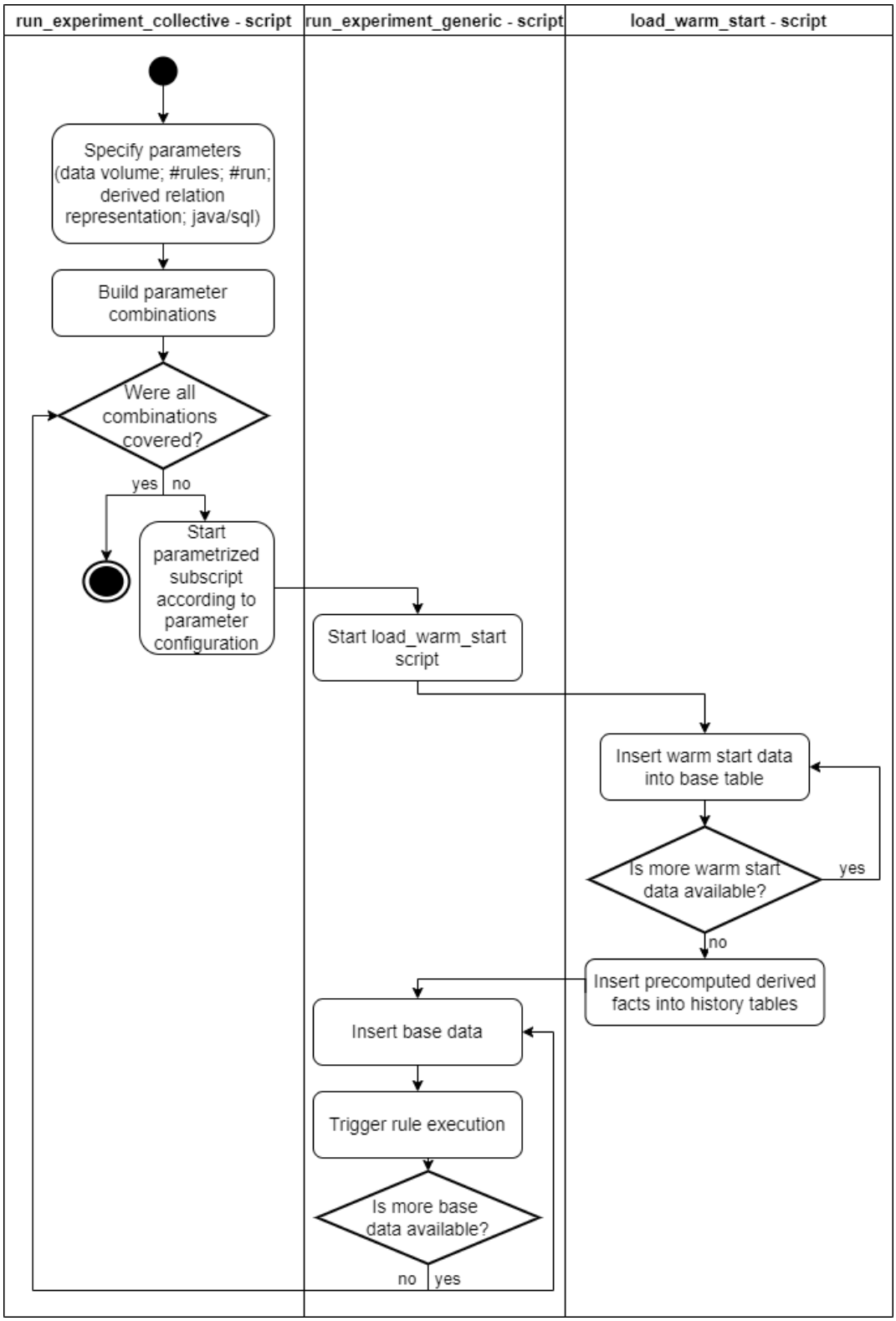


Figure 26: Separation of tasks concerning experiment execution into scripts.

When investigating a table-based approach, fact derivation is triggered differently when compared to a view-based approach. Derivation of facts for views is triggered at query time, whereas for tables it is already performed at insertion. Evaluating performance by tracking time consumed for finishing the corresponding *SELECT* or *INSERT INTO* command works similarly. However, a direct comparison between the two approaches in this way is not fair. Derived data is returned in the case of the view approach, but for the table approach an additional *SELECT* statement would be needed. To prevent the introduction of additional overhead by including a separate *SELECT* statement into the performance evaluation for the table approach, the *RETURNING* clause is added to *INSERT INTO* statements. By using this setting, the difference in performed workload is reduced to the materialization of derived facts. Finally, time measurements are collected in structured log files.

The performance results highlighted in Sections 10.2 and 0 are based on different system configurations, which were adapted iteratively. Different system configurations and resulting performance impacts are covered in Section 10.4.

10.2 Handling of derived relations and rule implementations

This section focuses on performance impacts of concrete differences in terms of handling derived relations with views, materialized views or tables, and rule implementations. To maintain high comparability within individual experiments, configuration parameters that are not associated with representing derived relations or rule implementations, are fixed within an experiment.

10.2.1 Experiments for Rule 1

For *Rule 1*, four different table-based approaches are described in this section. Two of those approaches use the previously described Java extension for PostgreSQL (*PL/java*), whereas the others use only plain SQL. Additionally, performance differences between using materialized views and a table-based approach are shown. Performance measures of using views are not covered in this section. From a use case perspective, forming the cross product of all flights and airports and calculating the distance does not yet produce meaningful information for an air traffic controller. However, the calculated distance could be used as a basis for further fact derivations, as, for example, determining the closest airport for each flight. Using views would require a repeated derivation of those distance facts, consuming additional resources and time. Therefore, materializing derived facts is considered a requirement for *Rule 1*. The performance of using views is investigated in experiments for *Rule 2*.

Figure 27 shows the performance difference of the four different table-based approaches. The approaches are differentiated according to the number of derived attributes per tuple, previously described in Section 8.4, and also whether Java is used for fact derivation. Performance is measured following varying workload combinations. In terms of *Data volume*, *Set03* (10 airports) and *Set04* (1 airport) are covered. The experiment is performed on a hardware setup of 16 cores. Tables are *unlogged*, and PostgreSQL is running in-memory on a RAM-disk. Time consumed for inserting data is not included in time measurements. The time limit indicates the theoretical maximum available time to conclude fact derivation, as it is the highest frequency in which new data is expected. On the x-axis, workload combinations are indicated. When comparing the consumed time between the different workload combinations, the negative performance impact caused by using user-defined functions in Java becomes apparent. Java-based derivation of facts is worse for all workload combinations when compared to the equivalent SQL-based variant. The time consumption of the SQL-based variants of

CO3_20 in comparison to the Java-based variants in C04_1 is roughly equal. However, the total workload of CO3_20 can be considered as 200 times higher than in C04_1.

The reason for the considerably worse performance of Java variants is attributed to the necessary context switches to the Java virtual machine (JVM). In the specific implementation in this project, fact derivation is performed on the row level to closely simulate a similar working principle to Datalog rules. Therefore, the number of context switches is also proportional to the workload. An alternative would be to adapt the architecture to perform all the fact derivation in Java similarly to the way it would be implemented when using an external Java client. For example, calling a Java function that retrieves all flight data, derives facts, and finally inserts all derived facts again into a table. However, this approach comes with the major disadvantage of foregoing DBMS-specific optimization strategies and benefits while also hiding Java logic behind hard-to-maintain user-defined functions. Performance would then strongly depend on the specific implementation of the Java function. Such an evaluation would need to account for Java-specific design decisions and remains for future work.

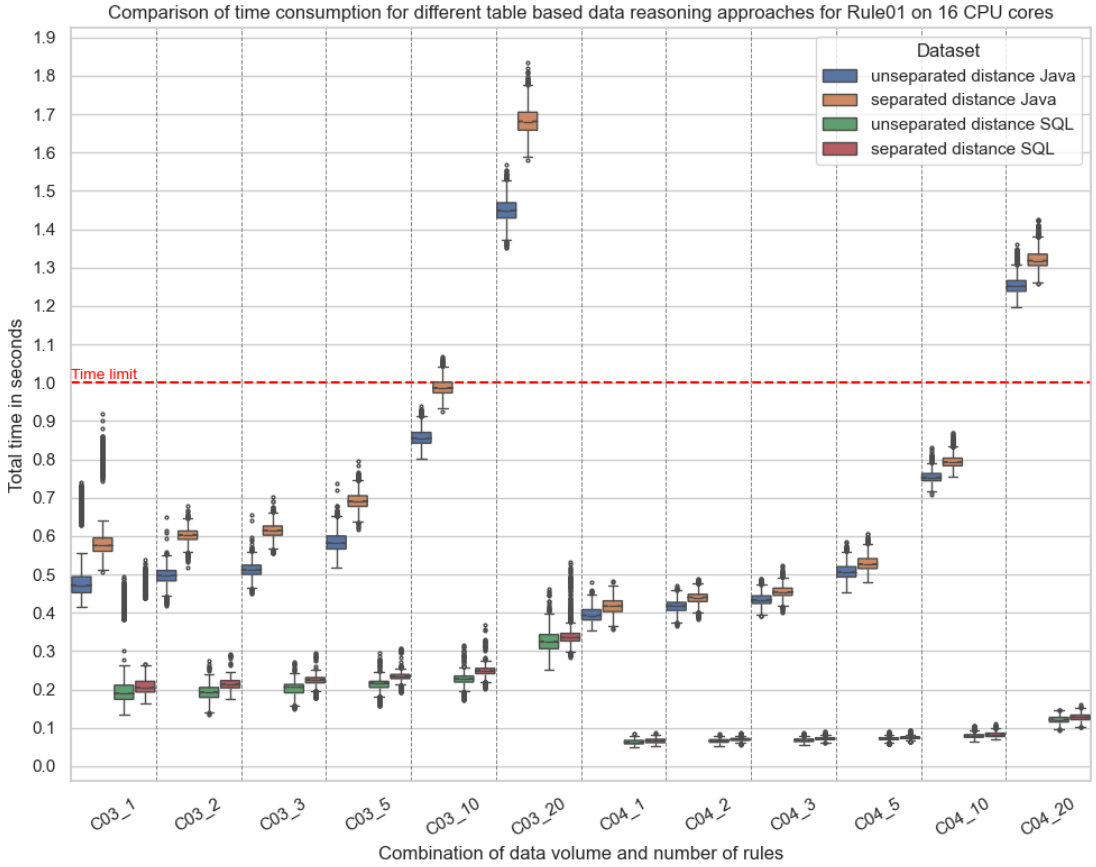


Figure 27: Performance differences for table-based querying and reasoning for Rule 1 on 16 cores

Figure 28 shows a comparison between using a table-based approach and a materialized view. The results are based on an early experiment that already showed worse performance when using materialized views compared to a purely table-based approach. The negative performance impact of

materialized views can be assigned to the additional refresh step needed when compared to a table approach. The refresh step was described in Section 8.3.1. Considering the worse performance as well as the increased architectural complexity, further performance evaluations for materialized views for *Rule 1* were halted, and the approach is considered impractical.

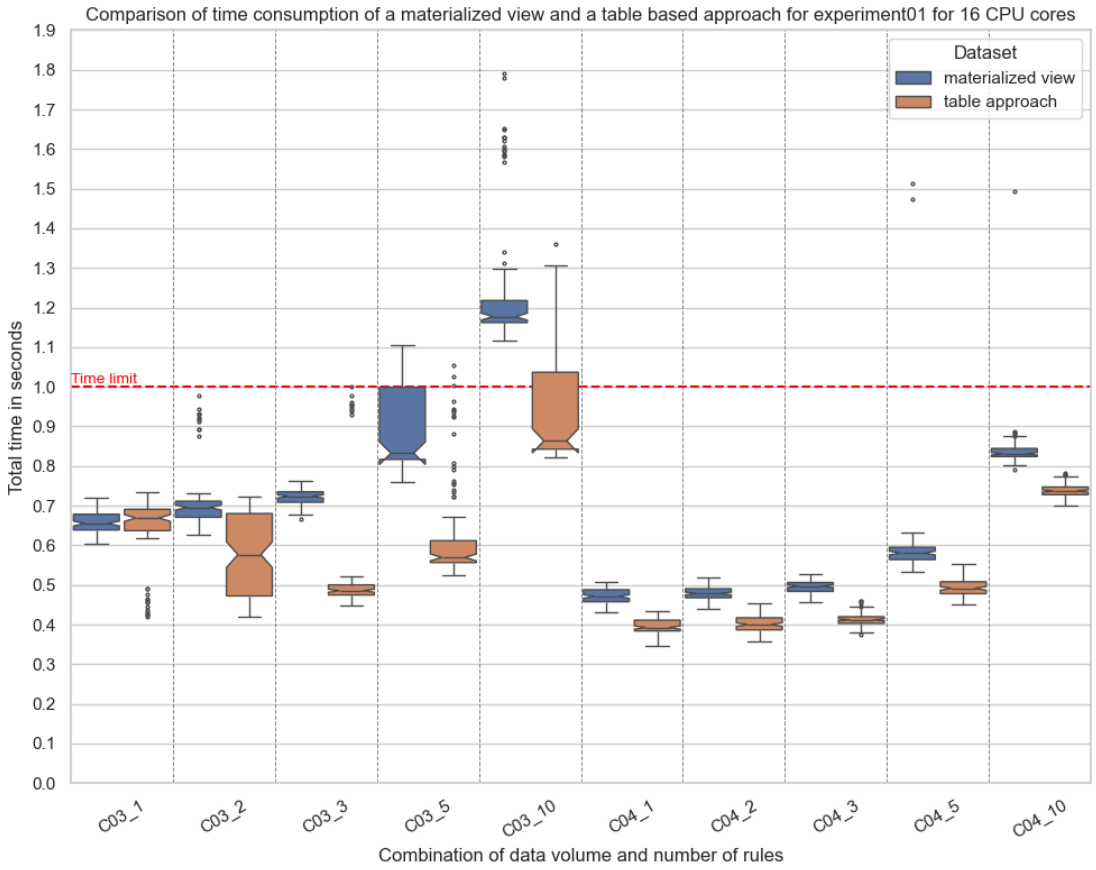


Figure 28: Performance differences for an approach based on materialized views and a table-based approach for Rule 1 on 16 cores. The figure is based on an early experiment and PostgreSQL configuration. It does not fulfill all final requirements.

10.2.2 Experiments for Rule 2

For *Rule 2*, a comparison between a table-based approach and a view-based approach is drawn. As the proposed rule predicts the position of all aircraft based on their current directional values, a use case where data is needed once at querying time is conceivable. For example, if the latest data is displayed as a trajectory on screen for an ATCO.

Similar to experiments for *Rule 1* highlighted performance measurements follow varying workload combinations. *Data volume* describes the number of flights associated with a timestep. Two different data volume sets are covered in this section. *Set02* concerns an increase of data volume to 10 times the originally captured flight data, whereas *Set03* represents the original data set. *Number of rules* covers again six different configurations from one to 20 duplicated rules.

Figure 29 shows the performance difference between four different configuration approaches. Two approaches use *tables*, whereas the other two use *views* for the *derived relation representation*. Settings are additionally differentiated on whether the fact derivation uses user-defined Java functions or relies solely on in-line SQL. On the x-axis, workload combinations are indicated. The dashed red line highlights the theoretical time limit imposed by the use case. In terms of data handling strategies, cron-job deletion is used for all table-based data sets. Time consumed for inserting data is not included in time measurements. Considering other configuration settings, the experiment is performed on 16 cores, unlogged tables, and with PostgreSQL located on a RAM-disk. When querying a view, data is derived and returned to the querying user afterwards. Deriving facts in a table-based approach would not behave in the same way, as data is not automatically returned to the user. To maintain comparability between table- and view-based approaches, the *RETURNING* keyword is used for the table-based approaches as described in Section 10.1. It can be shown that using pure in-line SQL over user-defined Java functions achieves a much better performance. Considering the combination *C02_20*, which concerns approximately 6000 inserted flights per timestep and 20 simultaneously executed rules per given timestep, for the SQL approach only around a fourth of the available time is used. In comparison, when using Java, both the table variant and the views variant exceed the available time limit. Reducing the executed rules to half also reduces the average time consumption by the Java-based approaches to close to under one second. However, some outliers exceed the time limit, which would render the approach unfeasible in a time-sensitive setting. Generally, the SQL-based approach is far more resilient to increasing workload. The reason for the worse-performing Java approaches is likely the necessary context switches to the JVM.

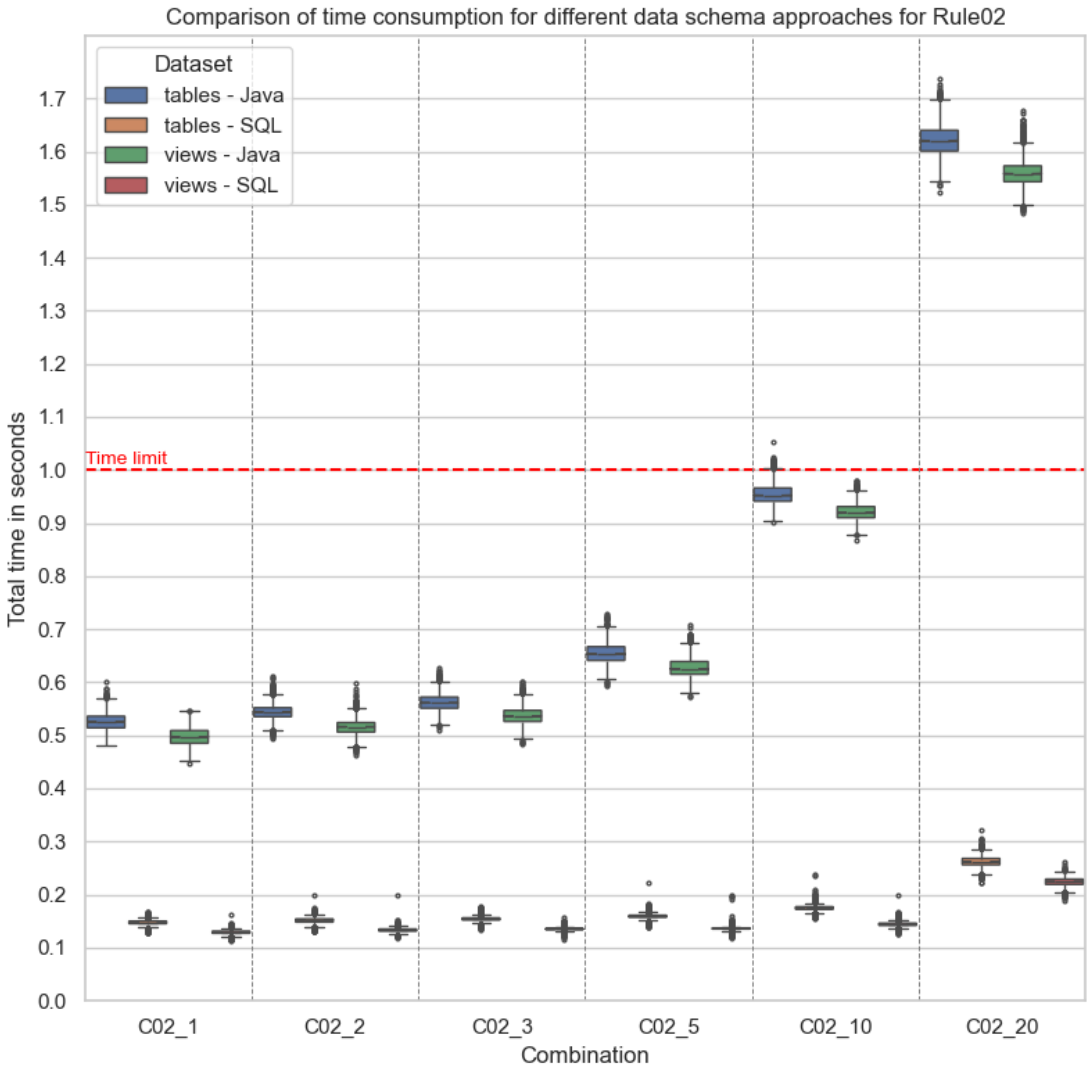


Figure 29: Performance differences between table and view approaches based on Java and in-line SQL.

10.3 Handling dynamic data

This section focuses on performance impacts attributed to used data handling strategies. Experiments highlighted in this section vary only in the data handling strategies within each experiment, while employing the same strategies in terms of *derived relation representation*, *rule implementations*, and other configuration parameters. This enables a fair comparison of varying data-handling approaches. Generally, four different data handling strategies were investigated.

Figure 30 shows the performance impact of different data handling strategies using tables as *derived relation representation* and user-defined Java functions as the *rule implementation* approach. This experiment was conducted on 16 cores, logging tables, and with PostgreSQL located on a RAM-disk. The same user-defined Java functions are used for fact derivation across all data-handling strategies. A total of 80 flights is inserted for each complete run of an investigated configuration, resulting in data equivalent to 800 flight graphs associated with each box plot. The two best-performing data-handling strategies are using *no deletion* and *cron-job deletion*. The *no deletion* strategy achieves only a slightly better performance than the *cron-job deletion* strategy. Generally, using *trigger deletion* performs on average similarly as well. However, there is a considerable number of outliers performing worse. The worst-performing data-handling strategy is the *trigger bulk deletion*. The reason for the high number of worse-performing outliers is the high number of trigger invocations. Generally, a trigger is invoked after a rule has finished deriving and materializing facts. Rules are triggered once after each flight graph insertion, which as a result triggers one data handling procedure per rule. As the number of rules increases, frequent triggering of small data handling procedures puts a high strain on the system. Using *trigger bulk deletion* invokes the same number of triggers; therefore, it produces a similar overhead. However, most of the time the context switch to the data handling procedure is unnecessary, as the required data threshold for deletion is not yet reached. This approach wastes resources on overhead, only to in turn require more for the deletion procedure when the threshold is reached. Using a *no deletion* strategy is promising when performance considerations of a steadily increasing data set can be disregarded. If this approach is not an option, for example in a productive zero-downtime environment, the *cron-job deletion* should be considered. In comparison to *trigger deletion*, the cron-job performs a single data handling procedure once per minute. This data handling procedure is decoupled from fact derivation and may handle an arbitrary number of data handling tasks in the database. In this experiment the scope of the cron-job spans over all historical tables for fact derivation as well as the base table. Requiring only a single context switch for data handling improves performance.

Co-funded by the European Union

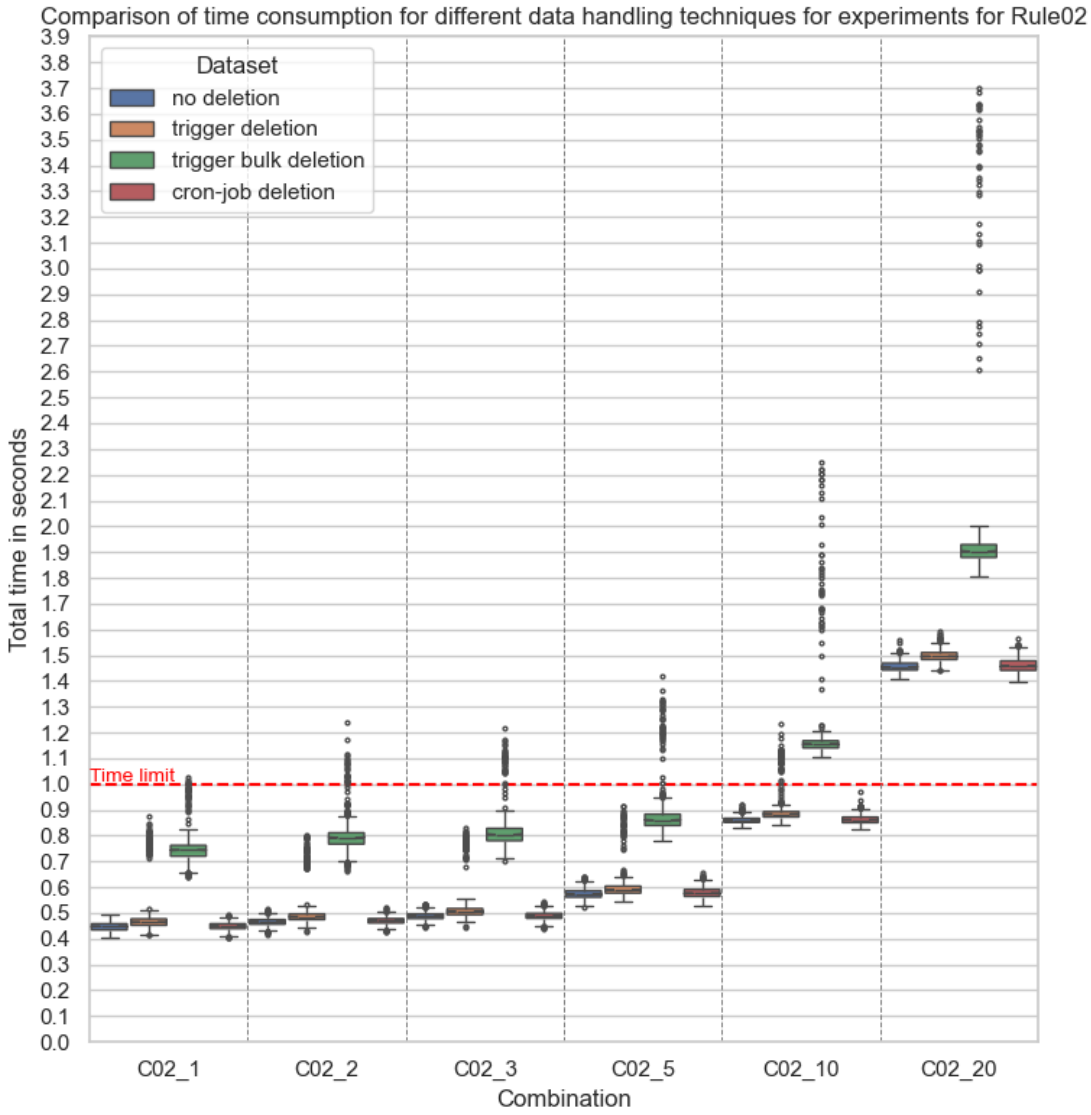


Figure 30: Performance differences between data handling approaches. 80 flight graphs are inserted per experiment run for each configuration. Resulting in data equivalent to 800 inserted flight graphs per box plot.

Performing *no deletion* or a *cron-job deletion* comes with the disadvantage of a higher volume of stale data. Given the results shown in Figure 30, the volume of stale data that accumulates during the runtime of the experiment does not impact the performance of the system in a meaningful way. However, the concrete volume of data that will be accumulated during simulation or when deployed in a productive environment is uncertain. Therefore, for this experiment, 319 flight graphs are inserted to show the impact of longer-running operations. Figure 31 shows the difference between using the two data handling approaches for the same system configuration. Measured performance data is split into three parts. The first part holds performance measures for the first 106 inserted flight graphs. At this point only data used to simulate a warm start is saved in the database. The third part shows performance data associated with the last 106 flights. As the *no deletion* approach does not perform

any data handling operations, stale warm start data and stale data of all preceding flights of the same run are saved in the historical tables. Substantial performance differences between a *no deletion* and *cron-job deletion* approach are not observed. In summary, at this level of workload, it does not matter which approach is chosen. However, as the negative performance impact of *cron-job deletion* is negligible, it is the safer variant to choose for long-running simulations.

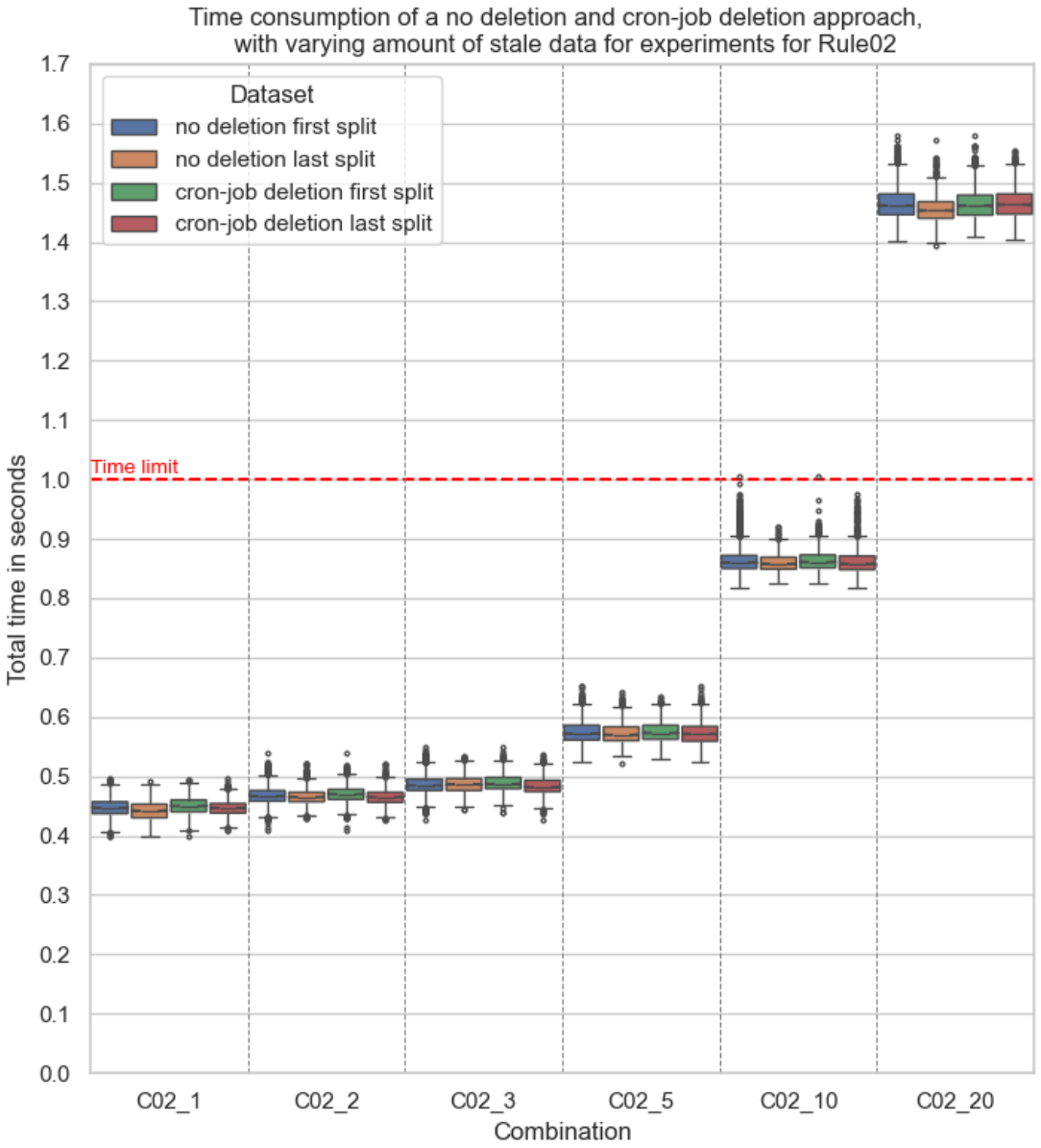


Figure 31: Performance differences between no deletion and cron-job deletion data handling approaches. Each box plot represents data associated with a split over the span of 10 runs, resulting in 1060 flight graphs.

10.4 Other configuration approaches

General achieved performance is additionally influenced by parameters not associated with the previously investigated areas of *derived relation representation*, *rule implementation*, or *data handling*. PostgreSQL comes with the option to fine-tune its configuration to achieve better performance. For example, the number of parallel workers available for tasks, logging levels and logging frequency, or available memory. Indexes on frequently used columns may also increase performance. PostgreSQL does not support a configuration that moves it entirely into memory. However, by moving PostgreSQL to a RAM-disk configured in the operating system, a full in-memory operation is achieved. Running PostgreSQL fully in-memory is vulnerable to data loss, for example, in case of a power outage. If the use case allows for in-memory operation, also using unlogged tables for further performance increases may be feasible. Finally, providing additional resources in terms of CPU cores may also improve performance.

10.4.1 Impact of indexation

Investigated rules *Rule 1* and *Rule 2*, which are described in Section 8.4, both focus on the latest available dynamic data (flights) of the base table. *Rule 1* additionally requires static airport data. However, in this case, all the available data is required for the cross product. Flight data is identified as the latest via the temporal context attribute $timestep$. Specifically, this is achieved by providing a *WHERE* clause in the *view* definition or *INSERT INTO* statement, which was shown in Section 8.4. An index on the temporal context of the base table increases the performance of the subquery in the *WHERE* clause.

Figure 32 shows a comparison between two experiments for *Rule 1*, where one uses an indexed temporal context and one does not. The results are based on an early experiment but show a positive impact of using an index on the *timestep* attribute. This increase in performance is more prevalent for experiments with higher data loads but can also be observed in those with only a single airport. The experiment shown is based on a configuration where PostgreSQL is on a RAM-disk, logging is enabled, the *RETURNING* clause is not used, and time consumed for data insert is not tracked. *Trigger deletion* is used as a data handling approach.

Co-funded by the European Union

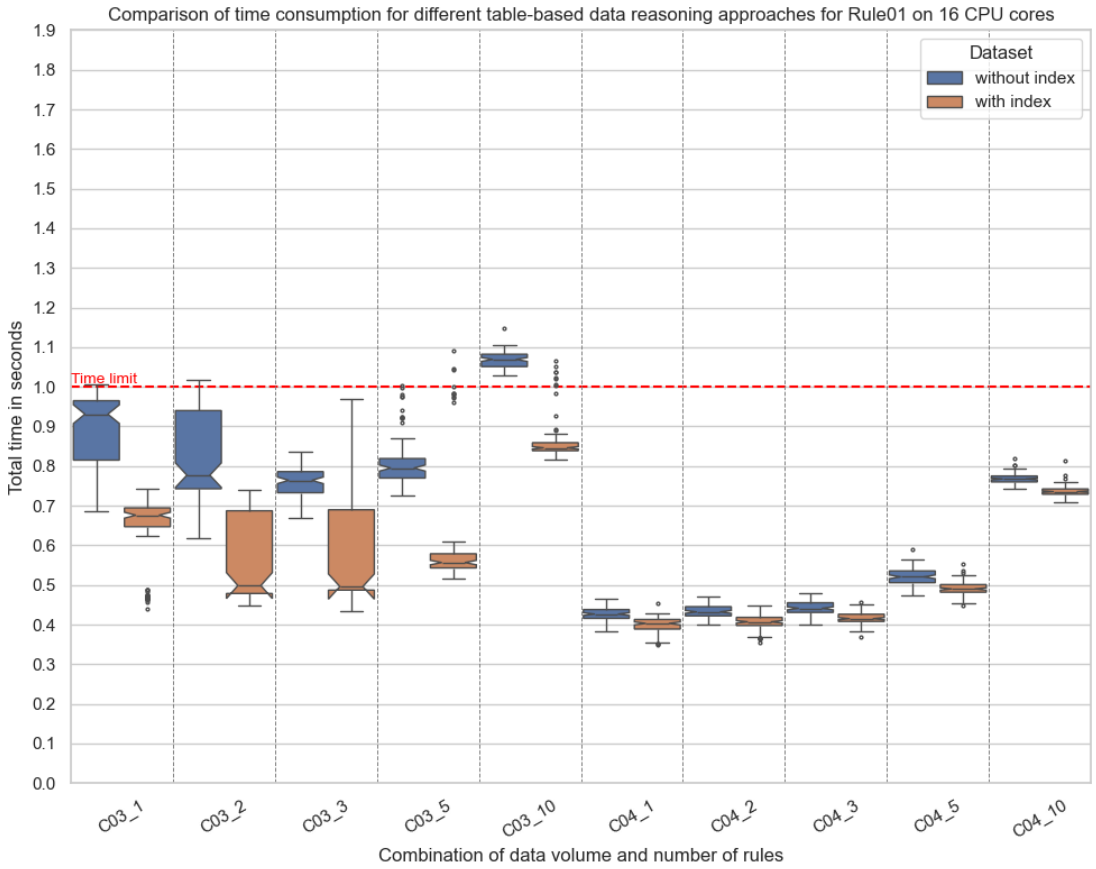


Figure 32: Performance differences between two table-based approaches for Rule 1. One utilizes an index on the temporal context attribute.

10.4.2 Impact of reducing fail-safety and treating the datastore as disposable

Performance can also be increased by reducing fail-safety and logging measures usually performed to guarantee a prevention of data loss. Disabling logging and moving the PostgreSQL instance into memory are two avenues that are explored to increase performance. PostgreSQL does not support full in-memory operation by default, which means some data is persisted to the disk eventually. In comparison, the alternative native-based RDF-based reasoning engine operates entirely in memory. Comparing two systems when one of them relies on disk I/Os and the other only performs operations in memory is not a fair comparison. Therefore, it was investigated whether PostgreSQL can be forced to calculate all derived facts without using costly disk I/Os. Two actions were set to achieve in-memory operation. First, the *shared buffer* was increased to 25GB. By choosing a high *shared buffer*, it essentially allows for caching of all available data during operation without the need to flush data to the disk. Secondly, PostgreSQL is moved to a RAM-disk. As the data used for fact derivation is at this point already cached in memory during the whole experiment, moving PostgreSQL to a RAM-disk achieves hardly any additional performance gain. However, as the whole cluster is moved to a RAM-disk, PostgreSQL-specific maintenance and logging tasks are also performed solely in-memory. The move to a RAM-disk comes with the major disadvantage of creating a vulnerable PostgreSQL instance, which should be considered disposable. A system crash or power outage would result in total data loss.

10.4.3 Impact of logging levels and unlogged tables

Considering the whole PostgreSQL cluster as a disposable data storage, log data normally used to rectify data corruption after a system crash is lost as well. Therefore, performing write-ahead logging (WAL) becomes unnecessary. PostgreSQL configuration allows for various settings controlling when to write updates physically to disk and ensuring crash safety via WAL [51]. Reducing crash safety and risking data corruption in a productive environment, in favor of increased performance, requires a conscious decision. This approach should only be used if other fail-safe measures are in place or the specific use case allows for it. For example, if data can be reconstructed from external sources or the PostgreSQL instance serves only as a read-only data store. If tables are marked as *UNLOGGED*, writing WAL is completely disabled, which further reduces fail-safety in favor of performance gain. *UNLOGGED* tables are automatically truncated after a crash or unclean shutdown, are not replicated to standby servers, and indexes created on them are also automatically *UNLOGGED* [52].

Figure 33 shows the performance difference between logged and *unlogged* tables if PostgreSQL is already established on a RAM-disk. Even though a slight performance difference can be seen in each workload combination, using *unlogged* tables in this case brings hardly any additional benefit. The experiment shown is based on a configuration where PostgreSQL is on a RAM-disk, logging is disabled, the *RETURNING* clause and insert time are not included, user-defined Java functions are used for fact derivation, and *cron-job deletion* is used as a data handling approach.

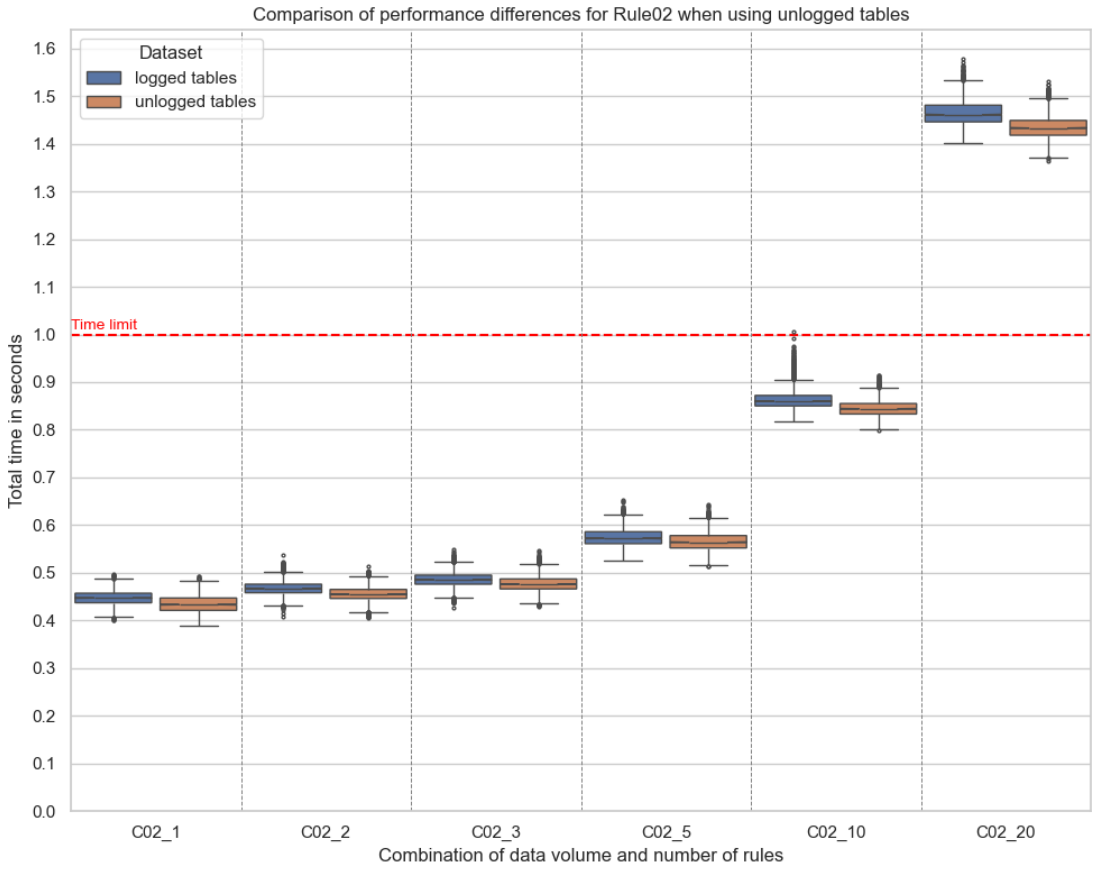


Figure 33: Performance differences between two table-based approaches for Rule 2 showing the difference of using logged and unlogged tables.

10.4.4 Impact of CPU cores

Experiments were performed on an 8-core and a 16-core CPU setup to investigate the impact of increased hardware resources. Figure 34 shows two different experiments for Rule 1, with their only difference being the available hardware. More available CPU cores do not always result in better performance. For a low number of rules, performance between the two approaches is roughly similar. However, for higher numbers of simultaneously triggered rules, the 16-core system performs visibly better than the 8-core system. The performance increase becomes prevalent when the number of rules exceeds the number of available cores. When more rules are triggered simultaneously than cores are available, full parallel fact derivation of all executed rules is not possible anymore, which has a visible impact on overall performance.

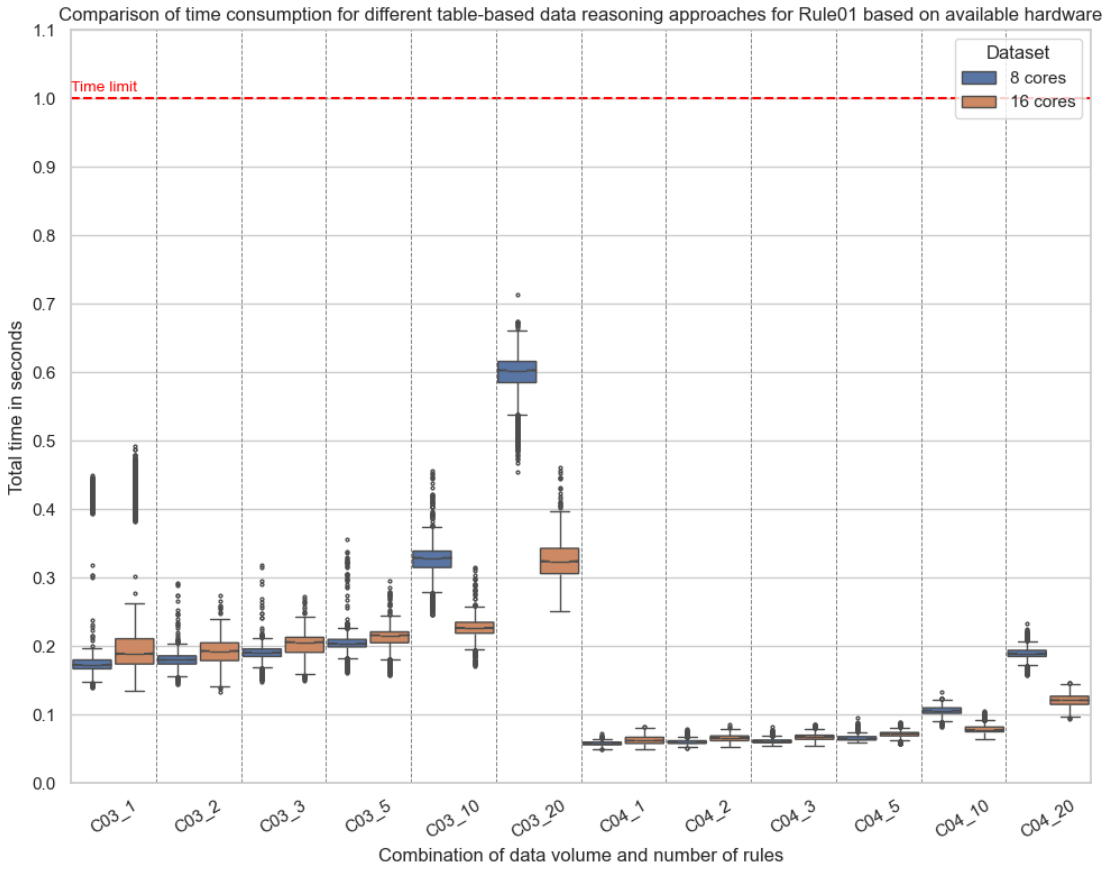


Figure 34: Performance differences between two table-based approaches for Rule 1. The first approach has 8 CPU cores available, whereas the second has 16 cores available.

A similar behavior can be observed for *Rule 2*. Generally, using views achieves a better performance compared to a table approach if the same hardware is used. Usage of views may not align with the requirements of the use case. Both view- and table-approaches show a similar behavior for *Rule 2* as observed for *Rule 1*. For a number of rules smaller than the number of available CPU cores, the performance impact of increasing the number of cores is negligible. However, for *C02_10* and *C02_20*, the performance gap between the two hardware configurations increases visibly.

In terms of overall configuration parameters, experiments shown in Figure 34 and Figure 35 derive facts without using Java, PostgreSQL runs on a RAM-disk, a *cron-job deletion* approach is used for data handling, tables are unlogged, *RETURNING* clause is used, but insert time is not included. For *Rule 1* fact derivation is performed in one combined distance attribute.

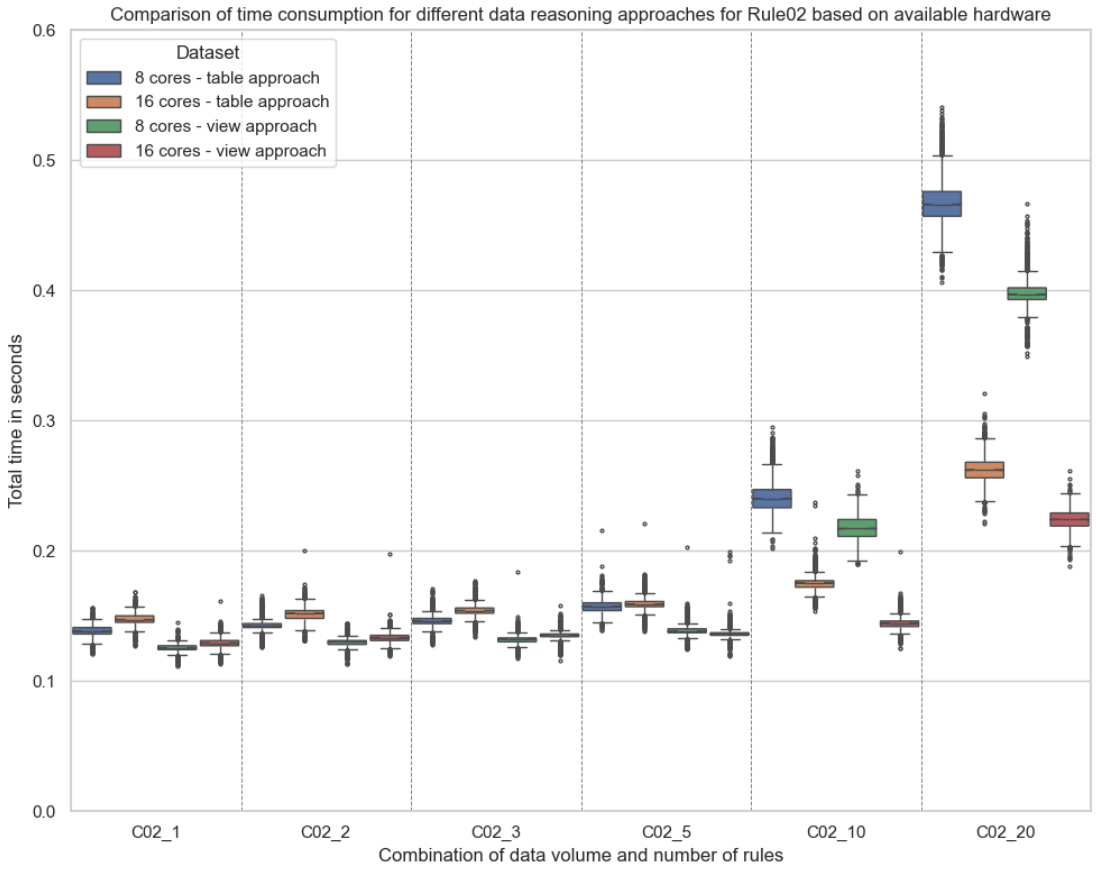


Figure 35: Performance differences between different reasoning approaches for Rule 2 based on available hardware.

10.4.5 Impact of automatic PostgreSQL optimization strategies

Rule 1 in the workload combination of *C03_1* shows an unusually high number of outliers. This agglomeration of outliers happens only for this specific workload combination and only for *Rule 1*. Figure 36 shows a snippet of Figure 34 visualizing specifically performance metrics for workload combinations *C03_1* and *C03_2*. Total workload is almost similar, its only difference being in one additional parallel rule execution for *C03_2*. However, outliers are vastly reduced, which is an unexpected behaviour. If a difference in performance occurs, *C03_2* should perform worse, considering the higher workload.

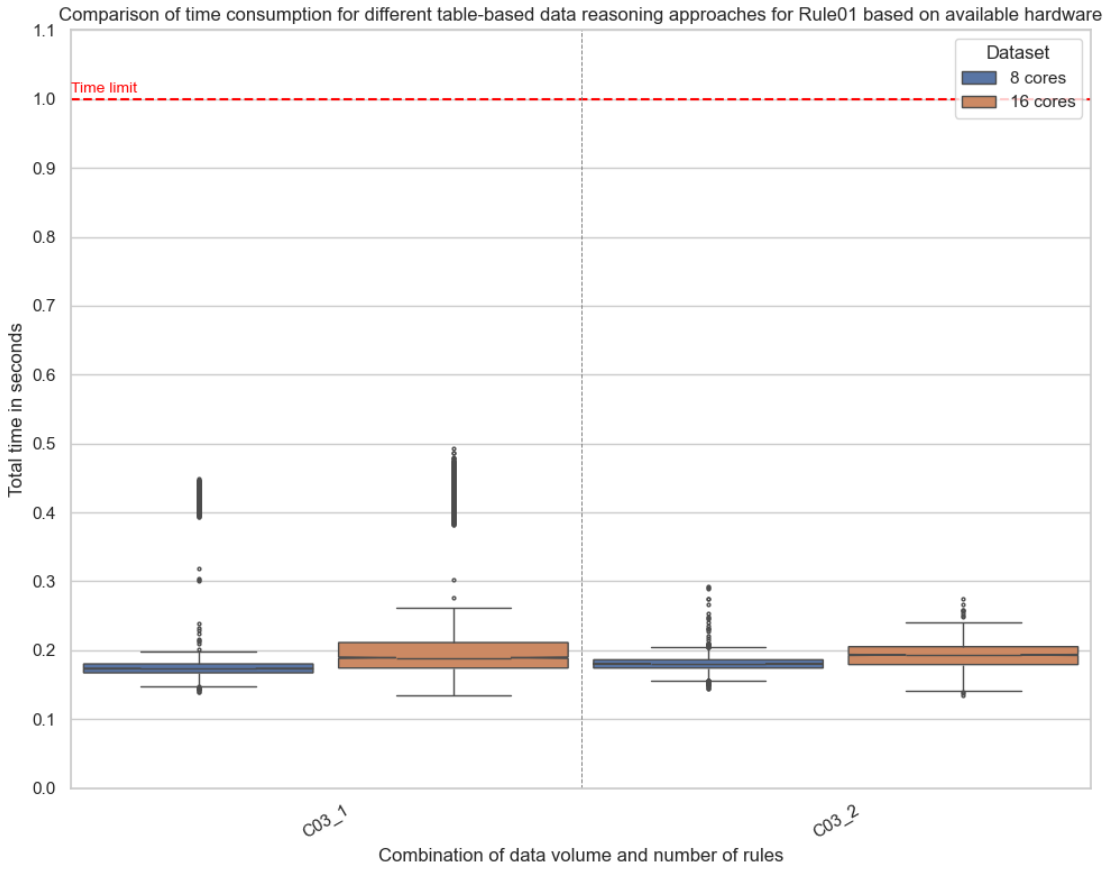


Figure 36: Investigation of the unexpected high number of outliers specifically for Rule 1 with workload combination C03_1.

The reason for this unexpected high number of outliers can be traced back to the way PostgreSQL specifically handles the join order in this case. When joining multiple tables, PostgreSQL automatically chooses the join order based on the most efficient order found by the query planner. In this work, only a maximum of two tables are joined in a single rule execution. Due to the low number of joins, the query planner is likely to find the best possible join order without forcing a specific join order by the developer [53]. For *Rule 1*, a cross-join between flights and airports is performed. The base table of flights has a comparably large number of tuples to the base table of airports. When executing the rule for a new timestep, PostgreSQL joins the tables via materializing the high-volume flight data in advance. However, with each timestep, the dynamic flight data changes completely while the airport data remains static. After a certain number of processed timesteps, the query planner switches the materialization order to perform the cross-join. This switch results in a visible performance increase. Figure 37 shows the unoptimized explain plan for a specific time step. In comparison, Figure 38 shows the optimized variant.

	Node	Timings	
		Exclusive	Inclusive
1.	→ Nested Loop Inner Join (actual=164.631..168.81 rows=5940 loops=1)	149.65 ms	168.81 ms
2.	→ Result (actual=0.082..0.084 rows=1 loops=1)	0.01 ms	0.084 ms
3.	→ Limit (actual=0.074..0.075 rows=1 loops=1)	0.003 ms	0.075 ms
4.	→ Index Only Scan using csv_flight_ts_idx on public.csv_flight as csv_flight (actual=0.071..0.0… **Index Cond**: (csv_flight.timestep IS NOT NULL)	0.073 ms	0.073 ms
5.	→ Seq Scan on public.csv_airport as p (actual=0.853..2.644 rows=10 loops=1)	2.644 ms	2.644 ms
6.	→ Materialize (actual=16.377..16.432 rows=594 loops=10)	0.033 ms	16.432 ms
7.	→ Index Scan using csv_flight_ts_idx on public.csv_flight as f (actual=163.764..163.989 rows=594 … **Filter**: (NOT f.is_on_ground) **Index Cond**: (f.timestep = $1) **Rows Removed by Filter**: 50	16.399 ms	16.399 ms

Figure 37: Unoptimized join order for execution of Rule 1.

	Node	Timings	
		Exclusive	Inclusive
1.	→ Nested Loop Inner Join (actual=0.712..3.87 rows=6010 loops=1)	3.204 ms	3.87 ms
2.	→ Result (actual=0.051..0.052 rows=1 loops=1)	0.002 ms	0.052 ms
3.	→ Limit (actual=0.05..0.05 rows=1 loops=1)	0.003 ms	0.05 ms
4.	→ Index Only Scan using csv_flight_ts_idx on public.csv_flight as csv_flight (actual=0.048..0.0… **Index Cond**: (csv_flight.timestep IS NOT NULL)	0.048 ms	0.048 ms
5.	→ Index Scan using csv_flight_ts_idx on public.csv_flight as f (actual=0.061..0.613 rows=601 loops=1) **Filter**: (NOT f.is_on_ground) **Index Cond**: (f.timestep = $1) **Rows Removed by Filter**: 49	0.613 ms	0.613 ms
6.	→ Materialize (actual=0.001..0.001 rows=10 loops=601)	0.001 ms	0.001 ms
7.	→ Seq Scan on public.csv_airport as p (actual=0.642..0.651 rows=10 loops=1)	0.002 ms	0.002 ms

Figure 38: Optimized join order for execution of Rule 1.

The join order switch does only occur in runs associated with the first workload combination. The reason for this lies with the structure of the testing scripts. A reset of the airport base table is performed at the beginning of each new test run. During a single run, the rest of the available parameters are iterated, and testing sub-scripts are invoked accordingly. Even though a specific workload for a single experiment might change due to an increasing number of rules, statistics and caches related to the airport base table remain available throughout a single run. The behaviour of reusing statistics and caches over multiple experiments does not invalidate overall performance metrics, as it is the expected behaviour of a warm start environment anyways. Ideally this behaviour would also occur in the first workload combination of a run, but a warm start is established via inserting precomputed derived data, which prevents PostgreSQL from creating necessary data statistics. An alternative would be to construct a warm start during runtime by starting from a cold start and only measuring performance as soon as a required threshold of available data is reached. However, this approach would unnecessarily increase testing time while still not guaranteeing the prevention of unoptimized joins. For better comparison with the native RDF-based implementation, unoptimized data can be easily excluded by comparing data associated with greater timesteps. Figure 39 shows a comparison of the performance impact between unoptimized and optimized joins for a snippet of the same data previously shown in Figure 36.

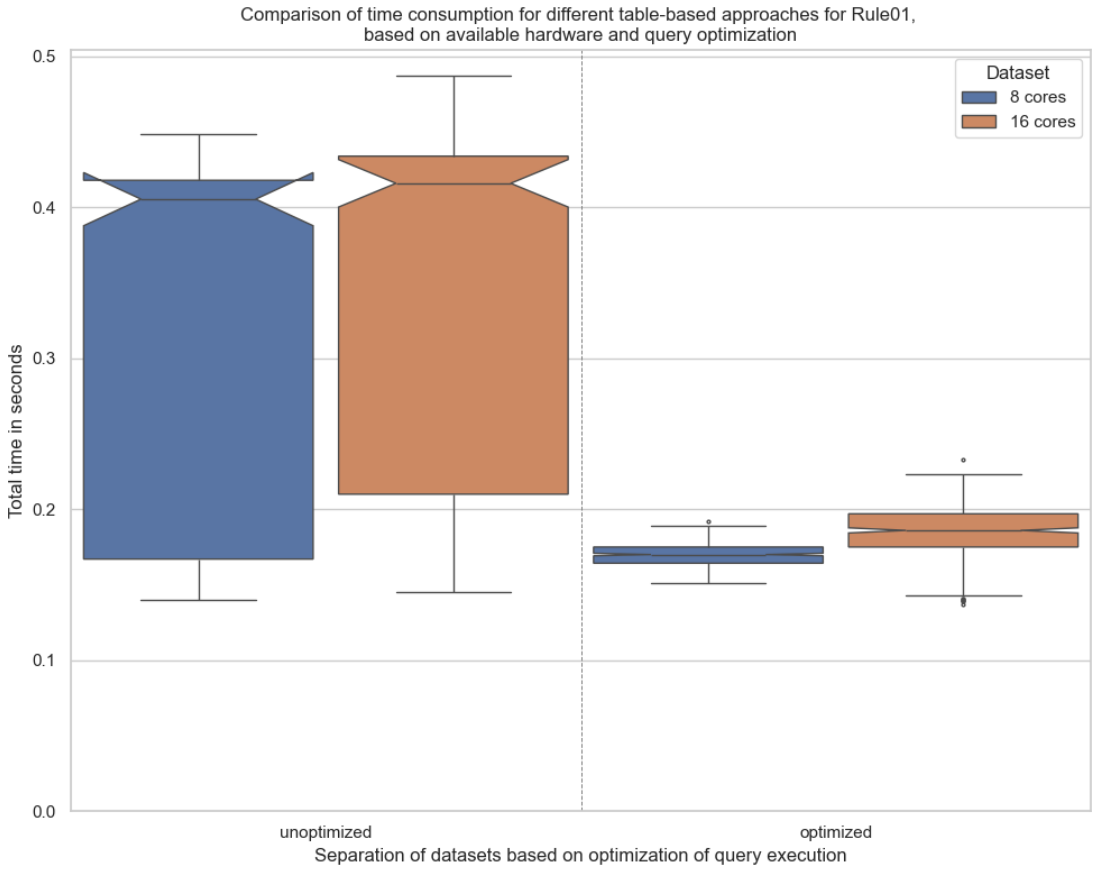


Figure 39: Comparison of performance for data splits associated with unoptimized and optimized cross joins.

11 Comparison of native RDF-based and relational KG management

In the following, we briefly describe the differences of the native RDF-based implementation and the relational implementation of the KG management system. We compare the two approaches in terms of architectural complexity and compatibility to other modules.

11.1 Architectural complexity

The architectural setup required for the implementation of the native RDF-based approach is more complex in some regards. The primary reason for this is the complexity of the additional external application, referred to as *proxy*. To ensure continuous operation, the proxy must always be adapted to the endpoints of the underlying RDF system whenever changes occur. Required effort arises from the numerous additional operations, such as *rule deletion*, *data deletion*, and *rule reinsertion*. Furthermore, rule generation must take place within the proxy due to the static nature of rule variants described in Section 5.3. In addition, two datastores must be operated in parallel, which increases the administrative overhead and, consequently, the overall complexity of the system. Changes to this system are difficult to implement because the individual architectural components are tightly interconnected. Modifications in one part of the architecture will very likely necessitate changes in other parts of the system. However, in terms of system maintenance, the effort remains manageable. This is because well-established and widely used Java classes are employed. Moreover, since Spring Boot was chosen as the foundation for the proxy and is well-maintained, the maintenance effort in this regard is also reduced.

For the relational approach, most of the architectural complexity is dependent on the data schema, and necessary maintenance efforts are reduced to standard database administration tasks. When changes to base data or rules occur, the underlying data schema may have to be adjusted accordingly. For the system developed in this work, most complexity and maintenance effort stems from a task scheduling perspective. For example, when using a table-based approach, triggering fact derivation via rules may be performed inside PostgreSQL by triggers, in a time interval by a cron-job or by external client applications. However, evaluating potential increases in complexity and necessary maintenance introduced by future client applications is dependent on those applications and, at this point, not feasible. Alternatively, when the use case allows for the use of views, scheduling efforts can be reduced by representing rules via views instead of tables, as it was described in Section 8.3. Furthermore, the relational approach utilizes two PostgreSQL extensions, *PL/Java* and *pg_cron*, which are open source and actively developed and maintained. Both are considered to introduce little additional complexity and required maintenance.

Co-funded by the European Union

11.2 Migration effort for existing modules

From a conceptual design perspective, the established KG system that was developed in AISA aligns more closely with the system using the native-based RDF reasoning engine than with the relational implementation. Reasoning rules were previously realized primarily as Java modules. However, using Prolog for the implementation of rules and performing the mapping from the KG to Prolog and vice versa was still an option.

Switching from the previously established KG system to a commercial native-based RDF reasoning engine requires rewriting all Java modules and all Prolog rules to rules in the proprietary syntax of the reasoning engine. In addition, architecture-dependent design decisions have to be considered. For example, depending on which of specific variant of the native RDF-based approach is chosen, the rule complexity or the complexity of the proxy that handles rules varies. However, the RDF reasoning engine is still based on an RDF KG, which is beneficial when external modules require an RDF representation of the KG.

Switching to the relational system instead generally allows for two approaches encompassing a different level of scope. First, a full switch from an RDF-based KG to the relational system may be performed, and the RDF KG is fully substituted by the relational system. Reasoning rules of Java modules can be partly reused due to the PL/Java extension for PostgreSQL. Prolog rules have to be fully rewritten to fit the relational system. However, if there are external modules and applications that rely on the RDF KG representation, those must be adapted accordingly as well. This may introduce additional high switching effort. If the first approach is impractical, a second approach relies on an exposure of the relational data layer as a conceptual KG. To achieve this, RDF views may be used if the DBMS supports them. PostgreSQL does not support the creation of RDF views. However, by using an OBDA framework, a conceptual representation as a KG can be achieved for the underlying relational data layer. Both RDF views and using an OBDA framework rely on an R2RML mapping, which must be defined separately and will increase switching effort.

11.3 Performance

Due to the restrictive nature of the license agreement of the commercial, native RDF-based implementation used in experimental development, this document does not publish performance data of the native RDF-based implementation. In any case, a direct comparison of the performance of the native RDF-based implementation and the relational implementation cannot be used as the sole basis for design decisions.

In general, the proposed relational implementation in this document trades flexibility for performance by relying on a fixed schema for the KG and by offering the possibility to implement an optimized set of SQL-based and externally represented (possibly Java-based) rules on a comparatively low level. However, any potential performance gain through this type of optimization comes at the expense of flexibility, both in terms of what type of information can be represented in the KG and in terms of queries that can be formulated.

12 Migrating from native RDF-based to relational KG management

While the native RDF-based approach may be used in the AWARE experiments, the relational approach could serve as the basis for a production system in the future. Therefore, this chapter demonstrates the migration from the native RDF-based implementation to a relational implementation, demonstrating the switching to the relational representation on a generic rule as well as the specific rules from the running example.

The rules implemented in the native approach follow a rule language closely aligned to Datalog. However, there are extensions and differences when compared to standard Datalog, which prevents an identical mapping to the one previously described in Section 6.4. In addition, a modified syntax is used in this work rather than a specific dialect from a commercial system.

12.1 Native rule structure and its mapping to a relational representation

Consider a rule $r1$ that is used to derive relations based on facts of a base relation R_B. For simplicity, R_B is dependent on a multidimensional but single-level context. However, each context dimension could be expanded to multiple levels, following the approach shown in Figure 21. Figure 40 shows $r1$ as a simple example based on generic relations and the modified syntax. Generally, the structure of $r1$ is separated into a rule head and a rule body. However, the rule body is additionally separated into two parts. One part concerns traversal of the graph and resolving variables, whereas the second part performs *number crunching* and string manipulation. In Figure 41, $r1$ is dissected into individual parts in more detail, and their connection to a named graph representation and relational representation is shown.

Equivalently to rule heads in Datalog, the rule head of $r1$ defines the derived relations. The rule $r1$ specifically defines two relations with schemas $R_{derived\_subject}(iri)$ and $R_{derived\_value}(iri, derived\_value)$. In an RDF graph representation, this would result in one triple $(iri - derived\_value - some\_value)$. The object $some\_value$ represents a concrete value of attribute $derived\_value$ in $R_{derived\_value}$. The subject iri represents a concrete value of attribute iri in $R_{derived\_subject}$ and in $R_{derived\_value}$. Additionally, $r1$ references an associated named graph for both relations. The named graph represents the context for the relations. The rule defines the context as a hard-coded reference to a variable named graph, in this case as a generic representation $derived\_relation\_named\_graph/context$, with one or more context dimensions. Resolving the reference to a specific named graph is not performed by the rule itself, but this specific native system uses the external proxy to dynamically resolve the references to named graphs, and in turn the context references of the defined relations.

For the relational representation, a type store approach is used, as was previously described in Sections 7.3.2 and 7.4. Therefore, multiple defined relations may be combined into a single relation $R_{derived}$ if, for a given tuple of $R_{derived}$, all attributes are dependent on the same context. The resulting relational representation is shown as a relational table but may also be implemented as a

Co-funded by the European Union

view or materialized view. Contextualization of $R_{derived}$ is achieved by context attributes $c_1, \ldots, c_n$. The derived relation $R_{derived}$ follows the schema $R_{derived}(derived\_subject, c_1, \ldots, c_n, derived\_value)$, with a primary key $(derived\_subject, c_1, \ldots, c_n)$. Contextualization of $R_{derived}$ is based on R_B. R_B follows the schema $R_B(id, c_1, \ldots, c_n, base\_value)$. The attribute id would be a natural key for a tuple in R_B if R_B was context independent. However, due to context dependency, the primary key of R_B is $(id, c_1, \ldots, c_n)$. Each context attribute is based in a single context dimension. In general, as described in Section 7.3.2, de-normalizing derived relations, for example to improve performance for dependent rules, is a valid approach as long as all attributes in the derived relation depend on the full context $(c_1, \ldots, c_n)$.

In the rule body, contextualization in the proprietary rule language follows a similar approach as in the rule head. Base relations are queried, and variable-dependent relations are defined similar to IDB relations in Datalog. The associated base relation for $r1$ is shown in a named graph representation in Figure 41. To perform number crunching and string manipulation, derived variables are defined by using function calls and arithmetic operations. The resulting values are bound to derived variables that may be defined in the rule head. For $r1$, a value in the base relation in *degrees* is transferred into its value in *radians*.

In the relational representation, the rule body is transformed into an *INSERT INTO* statement on the derived relation, similar to the approach of mapping standard Datalog to SQL shown in Section 6.4. The conversion from *degrees* to *radians* is mapped as shown in Figure 41. The string manipulation task is mapped similar to a derived variable. Assume that the concatenation of ":derived-subject" and id serves as a unique identifying IRI in the RDF-based approach. As described previously, in the relational approach, a unique primary key is established by $(id, c_1, \ldots, c_n)$. Therefore, constructing a unique IRI is not necessary, however the string concatenation is still performed to create values for id.

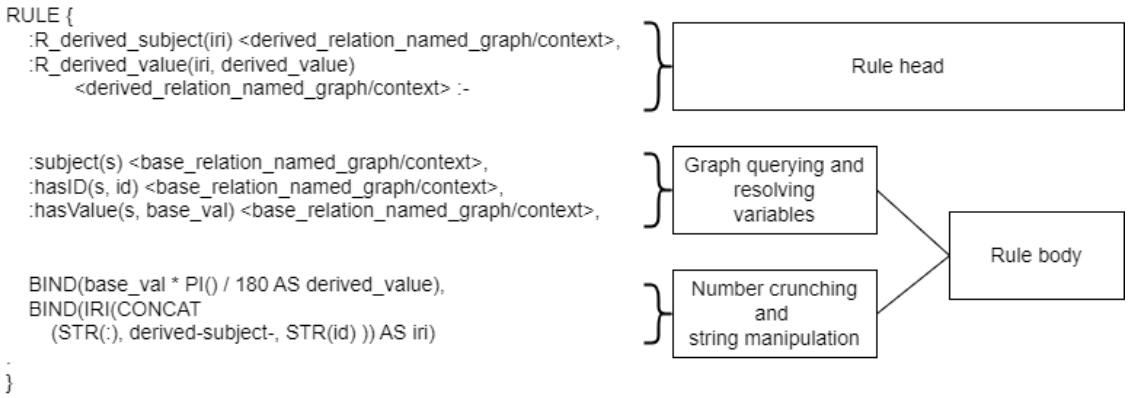


Figure 40: The structure of the implemented rule is separated into a rule head and a rule body. The first section of the rule body queries the graph of the base relation and resolves defined variables. The second section performs fact derivation by number crunching and string manipulation based on resolved variables and binds the results to new variables.

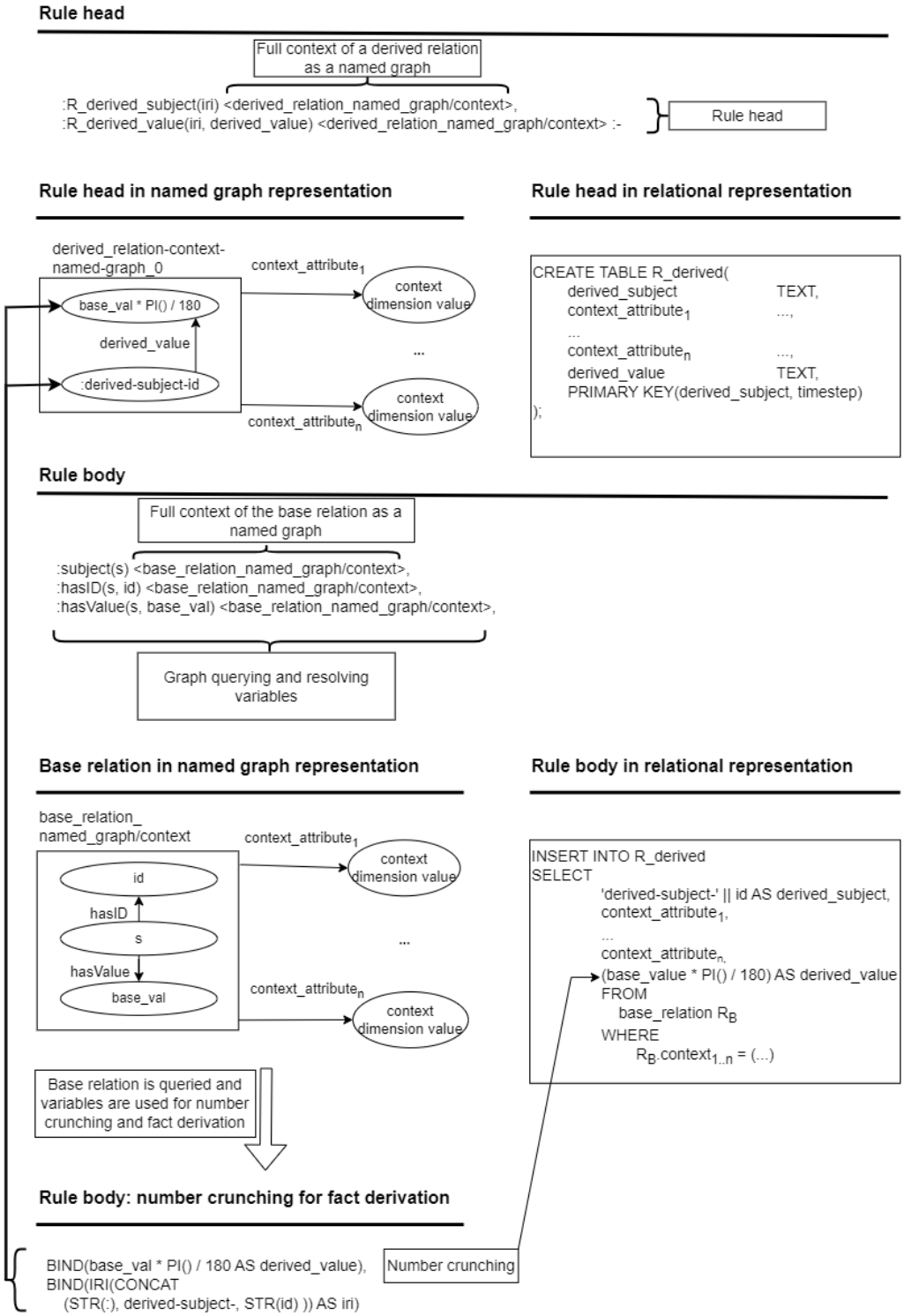


Figure 41: Mapping of a generic rule in the native system to a representation in the relational database. The representation of the derived relations in the rule as an RDF graph using named graphs for contextualization is shown. For the rule body, the base relation used for fact derivation is shown in a named graph representation.

12.2 Mapping Rule 1 to a relational representation

Figure 42 shows the mapping of individual parts of the implemented rule to counterparts in the relational representation. A generic mapping from the implemented rule to a relational representation was already shown in Section 12.1. In the rule head, the native rule defines six relations. As described in Section 7.4, for the relational representation, a contextualized type store approach is used. The six relations are combined into a single relation with a schema $R_{distance}(flightID, airportID, C_t, a_1, \ldots, a_n)$. The combination of the attributes $flightID$ and $airportID$ would pose a valid primary key for a context-independent variant of $R_{distance}$. Due to the context dependence, the context C_t is added to the primary key of $R_{distance}$. For this use case, C_t is based on a single-level, one-dimensional context of the time dimension, represented by $timestep$. $R_{distance}$ has the same context dependency as the base relation R_{flight}. *Rule 1* specifically derives facts based on the latest available data of the base relation. Therefore, the context reference of the base relation includes only $Max(timestep)$ as context value for the time dimension. In general, if the use case or rule requires it, using base data associated with other context dimensions or context dimension values is valid. *Number crunching* may be, for example, performed by using user-defined Java functions or pure SQL. Both a Java and an SQL approach were implemented and described in Section 8.4. The concrete derivation logic that is dependent on the chosen implementation variant is omitted in Figure 42. The concatenation of an IRI string performed in the native rule implementation is not required in a relational representation. The IRI serves as a unique identifier to the subject *AirportDistance*. In the relational approach, tuples are uniquely identified via their primary key.

Co-funded by the European Union

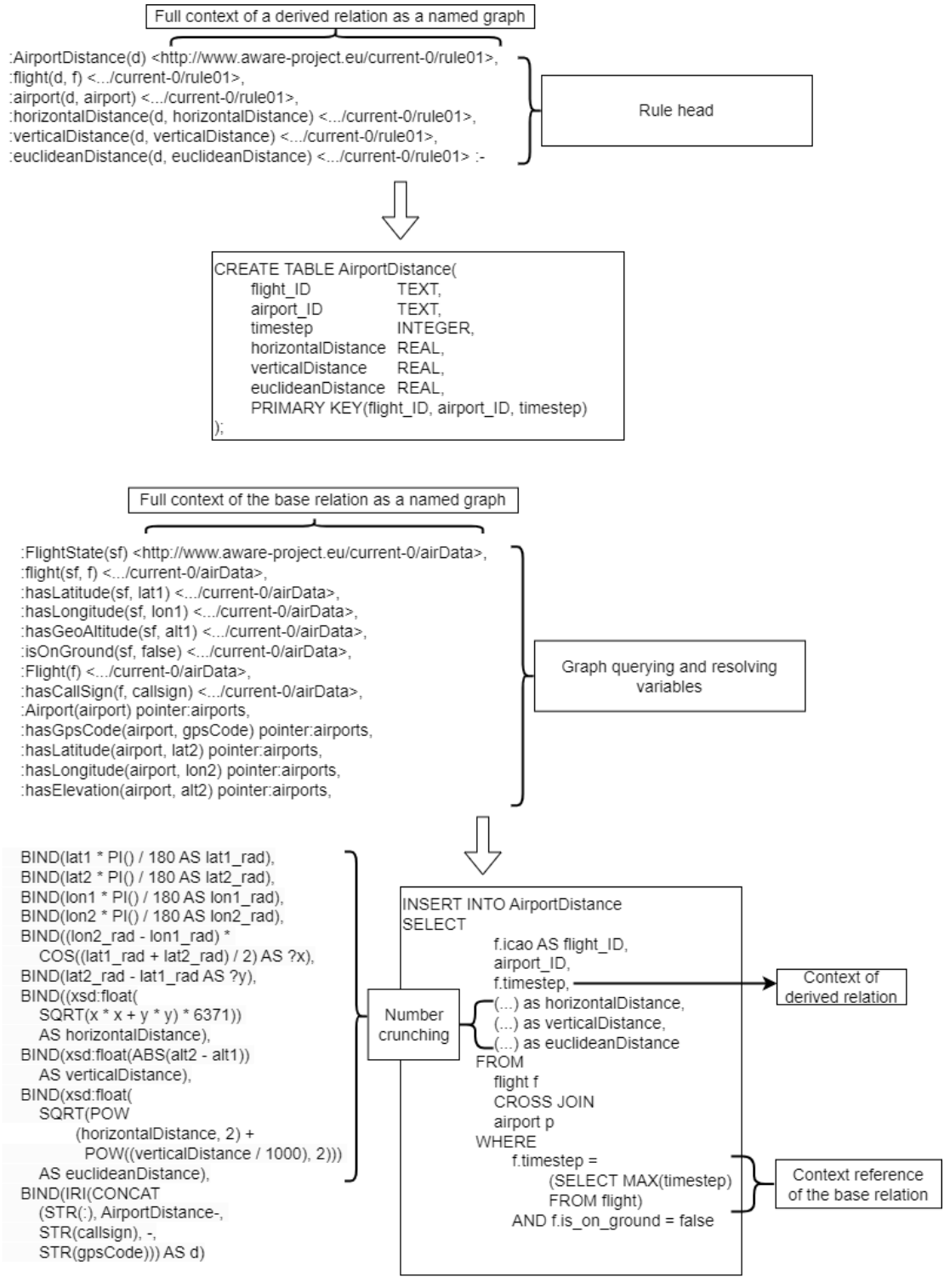


Figure 42: Mapping of individual parts of the rule implementation in the native RDF-based system to a representation in a relational database. The rule head defines the schema of the derived relation, and the rule body derives the facts. Contextualization is achieved by using named graphs. In this example, the rule explicitly specifies named graphs. In the native RDF-based system the specified named graphs are treated as references,

so that *current-0* of the specified named graph is, in this case, resolved to the named graph associated with the highest timestep.

12.3 Mapping Rule 2 to a relational representation

Figure 43 shows the mapping of individual parts of the implemented Rule 2 to counterparts in the relational representation. A generic mapping from the implemented rule to a relational representation was already shown in Section 12.1.

In the rule head, the native rule defines six relations. As described in Section 7.4, for the relational representation, a contextualized type store approach is used. The 6 relations are combined into a single relation with a schema $R_{predictedState}(flightID, C_t, a_1, \ldots, a_n)$. The attribute $flightID$ would pose a valid primary key for a context-independent variant of $R_{predictedState}$. However, due to the context dependence, the context C_t is added to the primary key of $R_{predictedState}$. For this use case, C_t is again based on a single-level, one-dimensional context of the time dimension, represented by $timestep$. $R_{predictedState}$ has the same context dependency as the base relation R_{flight}. Similar to *Rule 1*, *Rule 2* specifically derives facts based on the latest available data of the base relation. Therefore, the context reference of the base relation includes only $Max(timestep)$ as context value for the time dimension. In general, if the use case or rule requires it, using base data associated with other context dimensions or context dimension values is valid. Similar to *Rule 1*, *number crunching* for *Rule 2* may be, for example, performed by using user-defined Java functions or pure SQL. Both a Java and an SQL approach were implemented and described in Section 8.4. The concrete derivation logic that is dependent on the chosen implementation variant is again omitted in Figure 43. The concatenation of an IRI string performed in the native rule implementation is again not required in a relational representation, as the IRI just serves as a unique identifier to the subject *PredictedState*. In the relational approach, tuples are uniquely identified via their primary key.

Co-funded by the European Union

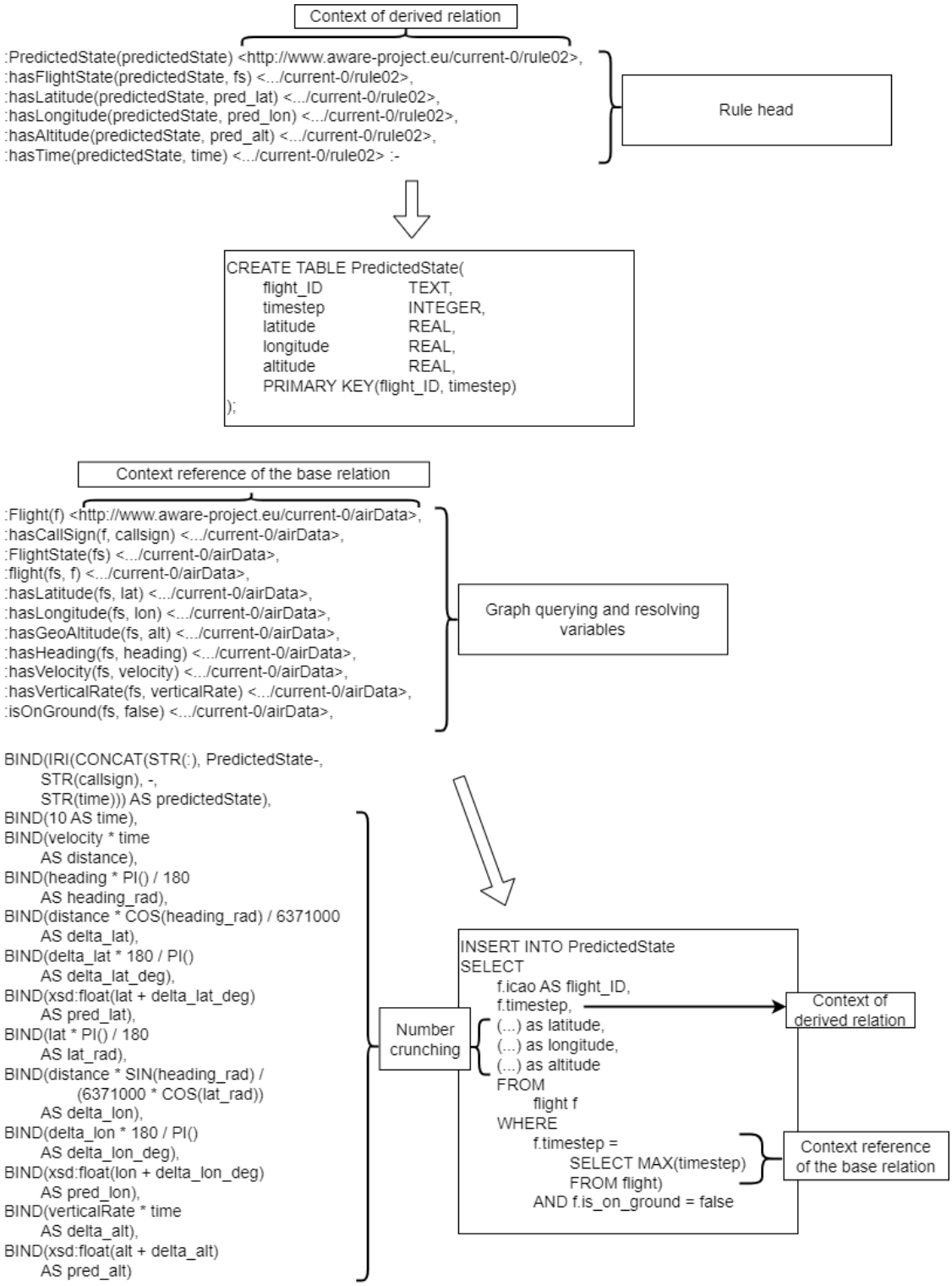


Figure 43: Mapping of individual parts of the rule implementation in the native RDF-based system to a representation in a relational database. The rule head defines the schema of the derived relation, and the rule body derives the facts. Contextualization is achieved by using named graphs. In this example, the rule explicitly specifies named graphs. In the native RDF-based system the specified named graphs are treated as references,

so that *current-0* of the specified named graph is, in this case, resolved to the named graph associated with the highest timestep.

13 Cloud-native lakehouse architecture for KG management

A knowledge graph (KG) represents a wide variety of facts about real-world entities, incorporating both instance data and ontological knowledge [81]. Organizations increasingly build their own KGs from diverse data sources, including relational databases and unstructured data, to gain a comprehensive view of their operations [100]. Their flexibility and adaptability to various tasks make KGs a potential unified foundation for business analytics and decision support [81], [93]. For example, in manufacturing, a KG can integrate information on production processes, order management, customer interactions, and supplier relationships, enabling strategic optimization of the supply chain. In air traffic management (ATM), where air traffic controllers and pilots rely on various sources to gain situational awareness of the air traffic network, (contextualized) KGs have been proposed as an integrated representation of the information required for decision support [83], [99], [54], [55].

Despite growing industry interest in KGs, research on KG data architectures suitable for business analytics and decision support remains limited. Existing KG data architectures and implementations focus primarily on data stores and query engines (cf. [57]). However, these platforms do not adequately address the *Three Vs* of big data—namely, volume, variety, and velocity [86]—which are critical for supporting applications in business analytics and decision support. Platforms such as Wikibase [75] and data stores such as GraphDB [91] and RDFox [89], [92] facilitate efficient management of monolithic, possibly contextualized KGs, but will nevertheless find a challenge in real-world settings where large volumes of a variety of data arrive at high velocity and have to be integrated into the KG, e.g., in the ATM domain. A proposed implementation of a KG system explicitly geared towards scalability for the purposes of analytics [95] primarily emphasizes querying while neglecting comprehensive architectural solutions for data ingestion. Those approaches could be used in combination with the data lakehouse architecture for KG management proposed in this chapter, which allows for rapid ingestion and on-demand KG construction for different applications and tasks.

The underlying assumption of the proposed approach is that many applications and tasks in business analytics and decision support do not require access to the entire KG at once. For example, in air traffic management, the pilot briefing for a flight from Munich to Vienna does not require information on the Spanish airspace. A contextualized KG (CKG), which partitions knowledge facts into multiple contexts of applicability of the represented knowledge [101], facilitates the selection of relevant knowledge facts. The architecture is explicitly designed to manage semantic data, producing contextualized RDF graphs structured as KG-OLAP cubes [98], which support reasoning, reporting, and other semantic analytics tasks. While the architecture ingests heterogeneous raw data, it produces contextualized RDF graphs structured by time, location, and topic dimensions, enabling semantic reasoning and analytics; this semantic orientation drives both indexing and task-specific KG construction throughout the system.

Modern data architectures for business analytics and decision support include *data warehouse*, *data lake*, and *data lakehouse* [80]. Data warehouses store structured, consistent data optimized for recurring and routine analyses, typically summarized in reports or dashboards. Data lakes archive raw, unprocessed data for more flexible, exploratory analyses and future use. However, the *schema-on-read* nature of the data lake requires complex cleaning and transformation, making access inefficient

for large datasets. Data lakehouses [63] combine the flexibility of data lakes with the structured nature of data warehouses by leveraging metadata, indexes, and caching to support complex analyses without sacrificing storage efficiency and query flexibility.

In this chapter, we propose the *KG Lakehouse*, a cloud-native data lakehouse architecture designed to ingest large volumes of heterogeneous, high-velocity data streams. While prior work such as [92], [87] have applied big data platforms to KG processing, our novelty lies in integrating KG-OLAP [98] into a cloud-native lakehouse architecture for contextualized, on-demand KG construction tailored to specific tasks. A key feature of the architecture is the on-demand construction of smaller CKGs that are tailored to support specific tasks in business analytics and decision support, e.g., in supply chain management or air traffic management. In this architecture, each ingested file is indexed and linked to a context derived from the file's content. This indexing mechanism facilitates efficient querying, filtering, and KG construction by maintaining semantic metadata about the stored raw data. The constructed CKGs are *KG-OLAP cubes* [98], which organize KGs into contexts. The contexts are hierarchically organized along multiple context dimensions, which can be used to aggregate the contained knowledge by merging different contexts. While our implementation builds on KG-OLAP cubes, the proposed architecture is modular and extensible, allowing for the integration of alternative CKG formalisms.

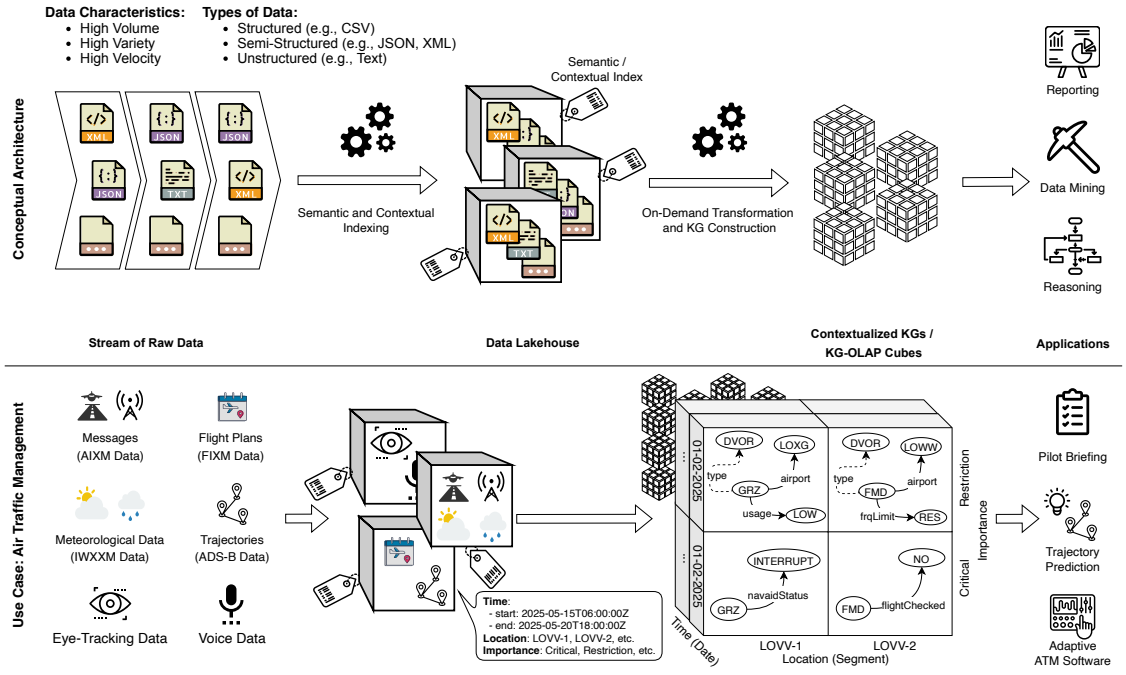


Figure 44: Overview of the proposed cloud-native data lakehouse architecture and an instantiation for air traffic management

Figure 44 in the upper part, provides a conceptual overview of the proposed *KG Lakehouse*, highlighting the main steps of the processing pipeline. The KG Lakehouse processes incoming high-velocity data streams, which may include a variety of structured, semi-structured, or unstructured files. The incoming files are ingested, the content of the ingested files analyzed to extract relevant semantic metadata about the context of the comprised information, e.g., time, location, topic, and importance, before the ingested raw files are stored and indexed in the data lakehouse. Rather than maintaining a

single, huge, monolithic CKG, with all its scalability issues [98], where most of the facts are anyway not needed by an application to perform a specific task, smaller CKGs containing only the contexts and facts relevant for a specific application and task, e.g., for reporting, data mining, or reasoning, are constructed on demand from the indexed raw files and supplied to the corresponding application.

We apply the proposed architecture in the ATM domain, although the architecture could also be deployed in other domains. We consider as prime examples: information filtering and summarization for pilot briefings [85], [97], trajectory prediction [78], and adaptive ATM software that provides air traffic controllers with the information required to perform their tasks. Figure 44, in the lower part, shows an example instantiation of the KG Lakehouse for the ATM domain. This KG Lakehouse receives continuous streams of messages about infrastructure, e.g., a temporary update of a navigation aid's usage type or frequency, updates of flight plans, weather reports, flight trajectories, eye-tracking of controllers to determine controllers' attention on specific tasks [72], and voice data, possibly as textual transcripts. The content of the incoming files is analyzed upon ingestion, to extract information regarding the semantics of the content, matching the files, for example, to certain time spans, locations (airspace segments, flight information regions, or airports), importance/criticality levels, or topics. A pilot briefing for a flight from Munich to Vienna on 14 May 2025 at 14:00 would require a different CKG than a flight on a different date or a flight connecting different cities. Furthermore, a pilot briefing requires a completely different CKG in terms of the topics represented compared to trajectory prediction or adaptive ATM software as proposed by the AISA and AWARE projects that considers the controllers' attention to display precisely the information that the controller currently requires to complete a task. Such an ATM software will frequently request CKGs with the required subset of knowledge facts valid at different times. This work follows the Design Science Research (DSR) methodology by Wieringa [87], focusing on the systematic design and evaluation of a design artifact—in our case, a scalable architecture for knowledge graph management built on top of a lakehouse infrastructure. The system was developed iteratively based on real-world requirements, grounded in the KG-OLAP framework [98], and evaluated using a realistic air traffic management scenario.

13.1 Related work

Knowledge-aware models leverage the unique characteristics of knowledge graphs (KGs), including their ability to integrate heterogeneous information and ontological knowledge. These characteristics have enabled a wide range of applications across domains such as information retrieval, question answering, social network analysis, deep learning, and analytics [110], [57]. However, traditional knowledge graph management systems (KGMS) often face challenges related to scalability, governance, and performance, particularly when handling high-velocity and diverse data streams. Our proposed architecture addresses these challenges by dynamically generating contextualized KGs tailored to specific applications, extending the KG-OLAP framework [98]. Unlike traditional OLAP, which aggregates numerical measures, KG-OLAP organizes KGs into multidimensional structures known as KG-OLAP cubes, storing knowledge facts identified by dimensions such as time, location, and importance.

13.1.1 Knowledge graph management systems

Existing KGMS provide the infrastructure for integrating, storing, and scaling KGs, enabling their application across diverse domains. Systems like Neo4j [90] and Blazegraph [70] specialize in handling graph data using property graphs and RDF models, respectively. These systems enable complex graph

analytics and semantic reasoning but are often constrained by static data representations. Distributed frameworks such as Apache Spark GraphX [61] and Amazon Neptune [58] address scalability through partitioning, distributed querying, and hybrid reasoning methods [81]. Despite these advancements, these systems lack the flexibility to dynamically generate KGs based on real-time user-defined queries, a key feature of the proposed KG Lakehouse architecture.

13.1.2 Cloud-native and lakehouse architectures

The increasing demand for scalable and modular architectures has driven the adoption of cloud-native designs, which leverage containerization and microservices to ensure flexibility, fault tolerance, and dynamic scaling. By adopting a cloud-native approach, the proposed architecture enables independent scaling of components such as ingestion, indexing, and querying, making it resilient to fluctuating data loads. Furthermore, the use of the lakehouse paradigm combines the flexibility of data lakes with the governance and performance of data warehouses [62]. This integration allows raw data to be ingested and stored in its native format while supporting schema-on-read queries, ensuring efficient data access and management. The Databricks Lakehouse Platform [74] exemplifies this hybrid approach by integrating metadata management, caching, and indexing to support complex analytics in multi-cloud environments. While Databricks focuses on tabular data, the proposed KG Lakehouse architecture extends this concept to support heterogeneous data sources, enabling the on-demand construction of contextualized KGs. Kona et al. [84] enhance this idea by combining Stardog [104] with Databricks [74] to add semantic reasoning, demonstrating the potential for hybrid systems in big data analytics.

13.1.3 Advancements in KG storage and analytics

Recent advancements in KG storage and analytics have introduced innovative frameworks addressing scalability, dynamic updates, and integration of diverse data types. GraphAr [87] is a storage scheme optimized for managing Labeled Property Graphs (LPGs) within data lakes. By leveraging Parquet, GraphAr accelerates operations like neighbor retrieval and label filtering. While GraphAr focuses on efficient graph data storage, the KG Lakehouse architecture emphasizes dynamic KG generation, extending beyond static storage to support application-specific requirements. Similarly, Stream2Graph [68] constructs dynamic KGs from streaming data, enabling real-time updates and online learning. However, while Stream2Graph excels in evolving data streams, the KG Lakehouse architecture incorporates advanced indexing and contextualization, ensuring efficient querying and retrieval for dynamic analytical tasks. Chen et al. [94] proposed a system for constructing knowledge graphs from unstructured text using deep active learning, thereby reducing the dependence on labeled data. Their method complements the KG Lakehouse architecture by providing advanced data extraction and learning capabilities that can be integrated into the ingestion pipeline for improved automation and adaptability. Several frameworks specifically address large-scale KG processing and analytics. P-OLAP [69] extends OLAP principles to graph analytics, leveraging Apache Hadoop and MapReduce [59] for scalability. SANSA [82] offers a unified semantic stack for processing RDF data at scale using Apache Spark, supporting SPARQL, reasoning, and machine learning. While SANSA is optimized for batch processing of static RDF datasets, our architecture focuses on real-time ingestion, contextual indexing, and on-demand construction of task-specific CKGs using KG-OLAP. These approaches are complementary: our system could integrate with SANSA for downstream KG processing, and we plan to explore this integration in future work. Trinity [102], by contrast, efficiently handles online and offline queries using distributed memory storage. While both frameworks are

powerful, they target specific formats or rely on static KG structures, in contrast to our dynamic, context-driven architecture.

13.1.4 Hybrid and domain-specific approaches

The reviewed works highlight significant advancements in KG storage, dynamic updates, and analytics. However, the KG Lakehouse architecture differentiates itself through its modularity, real-time ingestion capabilities, and dynamic KG generation. Unlike systems that emphasize static storage or static query structures, the proposed architecture supports diverse data types, achieves high ingestion rates, and ensures governance through its cloud-native lakehouse design, making it well-suited for high-velocity, data-rich environments. Hybrid approaches demonstrate the potential for combining graph reasoning with big data workflows. Wikibase [108] offers open-source infrastructure for managing pre-existing KGs but lacks support for generating task-specific contextualized KGs on demand. Compared to federated SPARQL [106], linked data fragments [106], or virtual KGs [109], our system supports context-driven subgraph materialization from raw data, using semantic indexes and on-demand processing. It does not rely on pre-existing triple stores or query federation but instead operates as an active KG synthesis engine. The Databricks Lakehouse Platform [74] and Stardog [104] illustrate hybrid architectures that combine semantic reasoning with multi-cloud analytics, supporting metadata management and governance zones. In contrast, OLAP-specific models for graphs, such as KG-OLAP [98] and P-OLAP [69], define safe aggregation over graph-structured data, which our system instantiates for contextualized KG construction. Gassauer-Fleissner et al. [77] further highlight how cloud-native platforms and graph databases can be combined for financial crime detection. These works show progress in KG storage, updates, and analytics. In contrast, the KG Lakehouse emphasizes modularity, real-time ingestion, and on-demand KG generation. Rather than relying on static storage or queries, it supports diverse data types, achieves high ingestion rates, and ensures governance through its cloud-native design—well-suited for high-velocity, data-rich environments.

13.1.5 Architecture

A typical KG management system (KGMS) is monolithic, designed to build and manage a single centralized KG. Although KGMS implementations can leverage cloud infrastructure and distributed computing for storage and processing, the KG itself grows as a single entity, complicating querying and management as scale and complexity increase. In dynamic and data-rich domains such as air traffic management (ATM), where continuous data streams are generated, analytical tasks often focus on specific subsets of the KG, e.g., specific geographic regions, time periods, or individual topics. For individual tasks, e.g., a pilot briefing for a particular flight, access to the monolithic KG for the entire airspace is not required. A modular approach with the generation of context-specific KGs then improves performance and scalability. Therefore, this work follows the Design Science Research (DSR) methodology, focusing on the systematic design and evaluation of an artifact—a scalable architecture for KG-OLAP built on top of a lakehouse infrastructure. The system was developed iteratively based on real-world requirements, grounded in the KG-OLAP framework [98], and evaluated using a realistic air traffic management scenario supported by public tooling and reproducible infrastructure. Figure 45 shows the main components and processing steps for the proposed *KG Lakehouse* architecture. Our contribution builds on the formal KG-OLAP model [98], extending it through a modular architecture with hierarchical context indexing and query-driven KG construction. The design reflects real-world constraints such as data velocity and heterogeneity, guiding key architectural trade-offs. The architecture reflects key lakehouse principles, including separation of storage and compute (via MinIO

and microservices), schema-on-read (during KG construction), and metadata-driven indexing (using context hierarchies in Cassandra); a detailed discussion of these principles is provided in an online appendix [67].

The *Surface Service* acts as the unified entry point, providing a RESTful API to handle interactions from users, applications, and data streams. In general, we distinguish between *data ingestion* and on-demand *KG construction*. The modular architecture with a separation of ingestion tasks and graph construction (query) tasks allows independent scaling of each component to meet fluctuating data volumes and processing demands. Thus, resource utilization is optimized by scaling only the necessary services, which allows efficient on-demand generation of contextualized KGs that overcome the limitations of a traditional monolithic KGMS. In the following, we describe the main components and processing steps for data ingestion and KG construction.

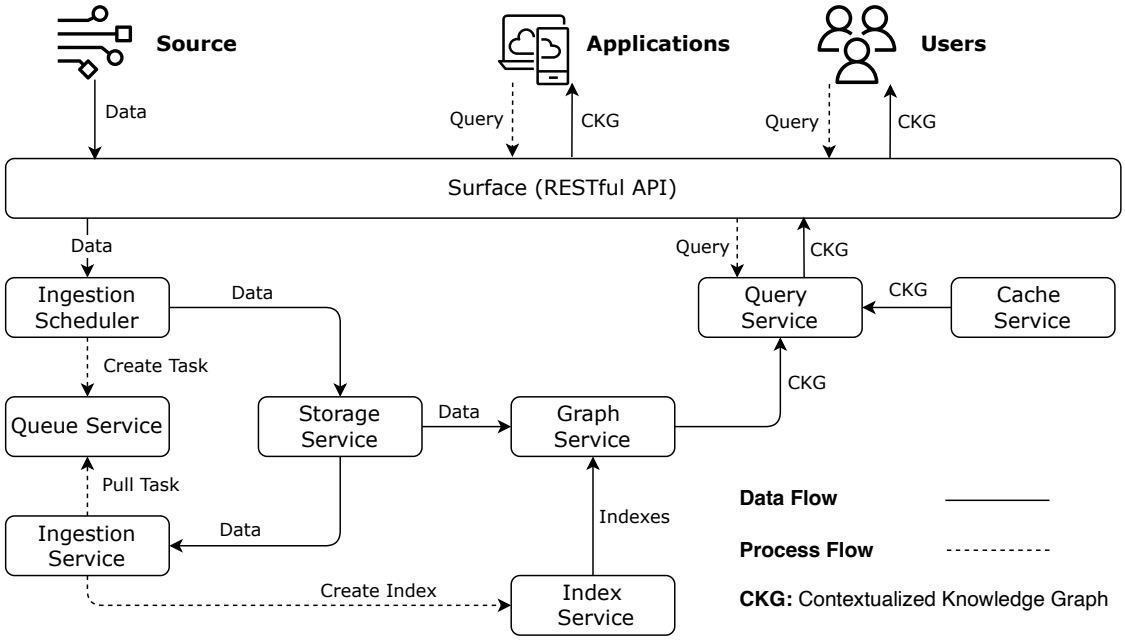


Figure 45: The proposed cloud-native data lakehouse architecture

13.1.6 Data ingestion

The KG Lakehouse employs an event-driven pipeline for scalable, resilient, and flexible data ingestion in cloud-native systems [103]. The *Ingestion Scheduler* and *Ingestion Service* manage the asynchronous data ingestion. The Ingestion Scheduler creates ingestion tasks, annotated with metadata. The metadata for a task consist of a description of the data source and content type, e.g., Aeronautical Information Exchange Model (AIXM) data, which are then used to select the appropriate domain-specific *Content Analyzer*. The raw files are persisted in the *Storage Service*, the ingestion tasks are pushed to a shared queue.

Figure 46 shows the high-level ingestion process conducted by the *Ingestion Service*. The Ingestion Service pulls tasks from the *Queue Service*, retrieves the data (raw file) associated with an ingestion task from the *Storage Service*, and analyzes the data using the appropriate content analyzers. The appropriate content analyzer is determined based on the metadata extracted during task creation, ensuring that the correct processing logic is applied for each data type. Domain-specific content analyzers designed to extract structured information from heterogeneous sources can be provided in a *plug-and-play* fashion. In this paper, we demonstrate the approach with a content analyzer for AIXM data, which represent messages about events in the air traffic network.

The content analyzers serve to annotate the ingested files with domain-specific contextual and semantic metadata. These metdata describe the *context* of validity for the information contained in the files. The context can be determined temporal intervals which the information is valid for, geographic locations that the information concerns, and the described topics, e.g., navigation aid restrictions, airspace closure or construction activity. Thus, the extracted context from a file is *multidimensional*. The context dimensions, in turn, are *hierarchical*. The contexts of a file are *upserted*—i.e., inserted or updated—into the *Index Store* to maintain accurate contextual indexes that can later be queried to retrieve files for KG construction.

Figure 47 illustrates the process of content analysis and context extraction using an example from the ATM domain. In this example, an AIXMBasicMessage is processed. This message follows the AIXM standard [56] for exchanging aeronautical information. The index service extracts information describing the *time* of validity (1 January 2018), the *location* with which the information is concerned (Vienna Airport or LOWW) and the *topic* (AirportHeliport). This information is hashed and stored, linking the ingested file to the derived context for efficient retrieval in the process of KG construction.

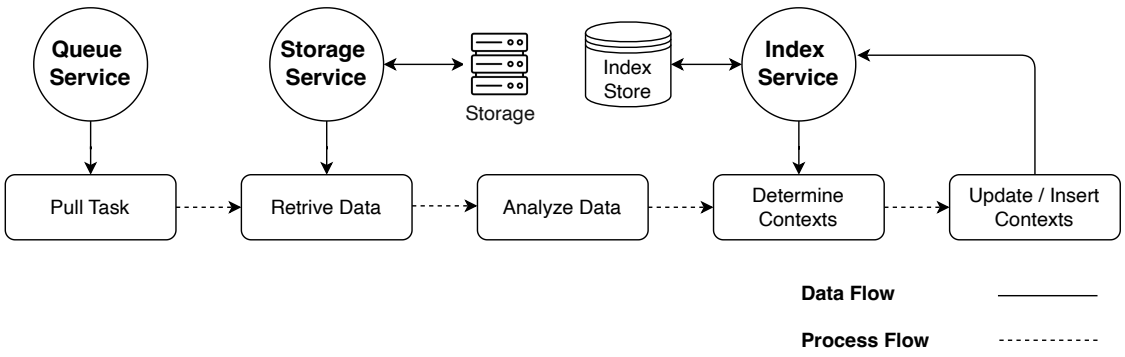


Figure 46: Overview and example of the data indexing process

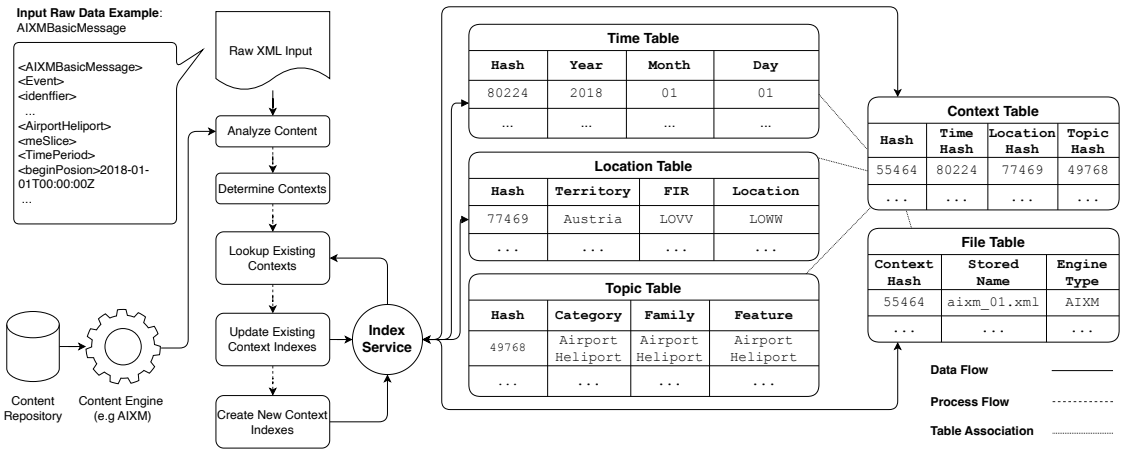


Figure 47: Overview and example of the data indexing process

13.2 Knowledge graph construction

The KG construction process retrieves relevant files for specific contexts on demand, based on a user-defined KG specification ("query"), and generates a contextualized KG (CKG) in form of a KG-OLAP cube. Figure 48 illustrates the KG construction process, detailing the interactions between *Query Service* and *Graph Service* in the construction of task-specific CKGs.

The KG construction process starts with the submission (via the Surface) of a KG specification to the *Query Service* by a user or application. The KG specification defines the temporal and spatial applicability as well as other dimensional attributes, e.g., the topic, of the desired CKG, depending on the configuration of the system and the target domain. For example, a request for *AirportHeliport* data for Austrian airspace during January 2018 would constrain the time to 2018 at the year level and January at the month level, the location to Austria, including relevant flight information regions (FIRs) and segments, and the topic to the *AirportHeliport* category.

The Query Service validates and parses a KG specification to extract context hierarchies (*SliceDiceContext*) and dimension mappings (*MergeLevels*). Specific contexts are directly derived from the specification, while general contexts are determined based on overlapping or shared dimensions across existing contexts. The service then iterates through all relevant combinations to identify the required contexts. The *Query Service* prepares queries for the required contexts and retrieves the file names for the contexts from the *Index Store*, ensuring both specific and general contexts are included. The *Graph Service* constructs the KG by retrieving raw data files from the *Storage Service*, extracting relevant triples of subject, predicate, and object for each context, and merging the triples into the contexts based on the dimension mappings to create a CKG. The final step is the aggregation of the results according to the KG specification, with caching mechanisms optimizing performance by retrieving pre-computed portions of the CKG, if available. This design instantiates the formal KG-OLAP model [98], where context dimensions form well-defined hierarchies that guarantee the safety of OLAP operations such as slicing, dicing, and merging.

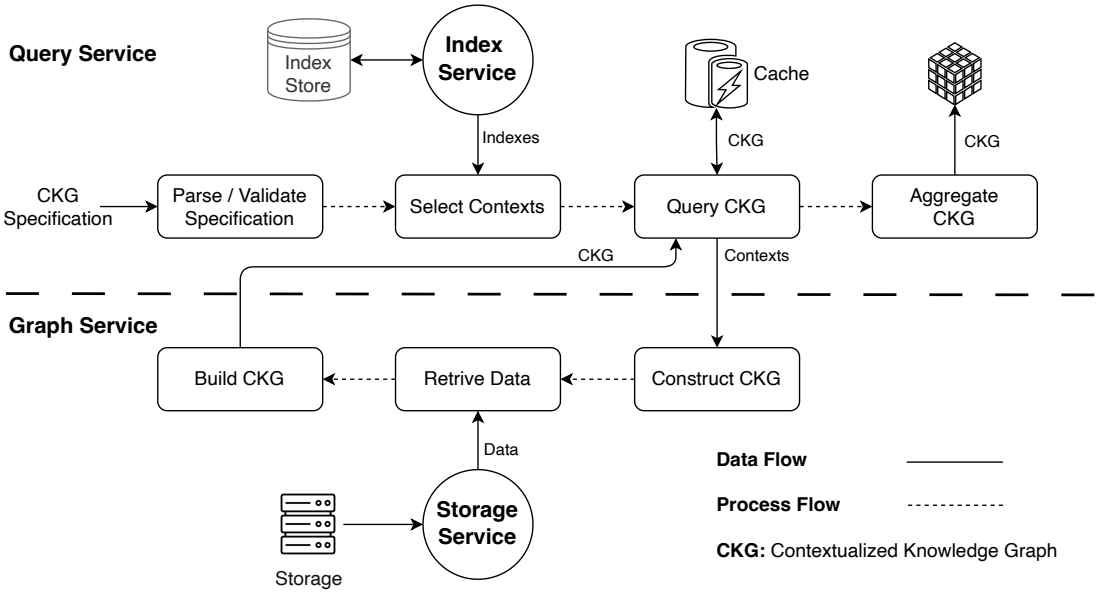


Figure 48: Overview of the KG construction process conducted by the Query Service and the Graph Service

13.3 Implementation

The proposed architecture follows a cloud-native microservice design, where the system is divided into small, independent components, enabling independent scaling of each component, allowing the system to efficiently adapt to varying loads. The prototype (available in an online repository [66]) is implemented in Java and deployed on a Kubernetes cluster. Components communicate asynchronously via events and remote procedure calls using gRPC [79]. Four core frameworks are required for deployment: **Cassandra** [71], **Redis** [96], **ElasticMQ** [76], and **MinIO** [68]. These services are implemented as *StatefulSet* services to ensure persistence and reliability. Each service can scale independently to accommodate system growth and, due to their stateful nature, each service supports quick recovery in case of failure. The architecture is flexible and allows the use of alternative technologies, e.g., **ClickHouse** [73] instead of **Cassandra** [71]. This design ensures that external services are interchangeable as long as they are cloud-deployable, distributed, and scalable. In our prototype, **Cassandra** serves as the index store, as it is a distributed NoSQL database well-suited for cloud-native applications and partitioned data. To ensure scalability, efficiency, and performance, we use modern object storage technology, specifically **MinIO** [88], which is cloud-native, high-performance, and optimized for large-scale file storage. An example configuration for deploying the system on Amazon AWS is provided in an online appendix [67]. The prototype is modular, cloud-deployable, and built with production-grade technologies (e.g., Cassandra, MinIO, Kubernetes). The prototype has processed millions of records in operational settings and includes complete infrastructure code with example AWS deployment.

13.4 Evaluation

In this section, we evaluate the scalability of our proposed architecture by measuring the performance of the key operations—namely, data ingestion and KG construction—under different loads. Hence, we conduct two series of experiments. The first series of experiments evaluates data ingestion using a

predefined load, denoted as **L0** (Table 12), which is repeatedly sent to the system. We vary the scale of data ingestion components deployed on the Kubernetes cluster to examine how increased scaling impacts ingestion speed. As expected from a cloud-native architecture, the results demonstrate that increasing the scale of ingestion components considerably improves data ingestion speed. The second series of experiments evaluates the system's ability to construct KGs of varying sizes on demand from different file loads stored and indexed by the system. The KGs (Table 13) are constructed from variously-sized datasets, from **L1** to **L5**, with multiple repeats of the KG construction process to observe how dataset size influences response time. For these experiments, the KG Lakehouse was deployed on a Kubernetes cluster [85] with eight nodes: two server nodes and six worker nodes. Each server node was configured with four virtual CPUs, 8 GB RAM, and 48 GB disk space. Each worker node was configured with eight virtual CPUs, 16 GB RAM, and 100 GB disk space. All nodes ran Ubuntu-22.04 Server (minimal) and k3s-1.24.6.

13.4.1 Datasets

We conduct experiments with realistic synthetic datasets from the ATM domain. Standard benchmark datasets lack the contextual and hierarchical structure required by KG-OLAP; therefore, we used a publicly available AIXM-based generator to produce realistic, reproducible test data. Instructions for reproducing the datasets are provided in the online appendix [65]. The raw files are generated using an AIXM data generator [64], which generates XML files according to the AIXM standard [56]. Each file represents a message (e.g., AIXM BasicMessage) and typically contains 20–100 RDF-equivalent triples. The files are organized along three context dimensions: *Location*, *Topic*, and *Time*. The *Location* dimension is hierarchically organized into territories, flight information regions (FIRs), and locations, where locations roll up to FIRs, and FIRs roll up to territories. The *Topic* dimension is hierarchically organized into topic categories, topic families, and topic features, where features roll up to families, and families roll up to categories. Finally, the *Time* dimension is hierarchically organized into year, month, and day. Days roll up to months, and months roll up to years. We considered three territories: Austria, Germany, and France. The Austrian territory has one FIR with three locations. The German territory has two FIRs, each with four locations. French territory has two FIRs, each with three locations. From the AIXM model, we considered five categories: **AirportHeliPort**, **Routes**, **Airspace**, **NavaidsPoints**, and **Procedure**. **AirportHeliPort** has two families, each with four features. **Routes** has two families, each with two features. **Airspace**, **NavaidsPoints**, and **Procedure** each have one family with one feature. We generated five datasets according to this structure, as shown in Table 12, which lists the number of files, quads, and contexts used for each dataset.

Load	Files	Quads	Contexts
L0	0.16M	3.18M	0.08M
L1	0.82M	15.92M	0.41M
L2	1.65M	31.84M	0.82M
L3	2.47M	47.76M	1.24M

Co-funded by the European Union

L4	3.79M	73.24M	1.89M
L5	4.61M	89.16M	2.31M

Table 12: Characteristics of the datasets used in the experiments

13.4.2 Baseline

To provide context for the evaluation, we conducted baseline experiments using a monolithic pipeline to ingest and construct KGs similar to the ones obtained from the KG Lakehouse. The pipeline, implemented in *Python*, for processing AIXM XML files, using the same dataset used in our prototype evaluation, and storing the resulting graph in Apache Fuseki [60], created a contextualized knowledge graph similar to the KG Lakehouse architecture. Fuseki was selected as it supports RDF and SPARQL; however, like other existing systems, it assumes a precomputed KG, limiting scalability. Our approach is orthogonal: it builds contextualized KGs on demand from raw data, avoiding these limits while remaining compatible with such systems. The pipeline was deployed on a single machine with eight virtual CPUs, 32 GB RAM, and a 100 GB SSD, running on Ubuntu-22.04 Server. While functional, the monolithic setup demonstrated significant limitations in scalability and performance. For example, processing **L0**, the smallest dataset, took considerably longer than the *lakehouse architecture*, despite leveraging multiprocessing for parallel XML processing. Resource monitoring revealed underutilized resources and bottlenecks in the monolithic graph construction and storage design, further emphasizing the scalability advantages of the distributed KG Lakehouse architecture.

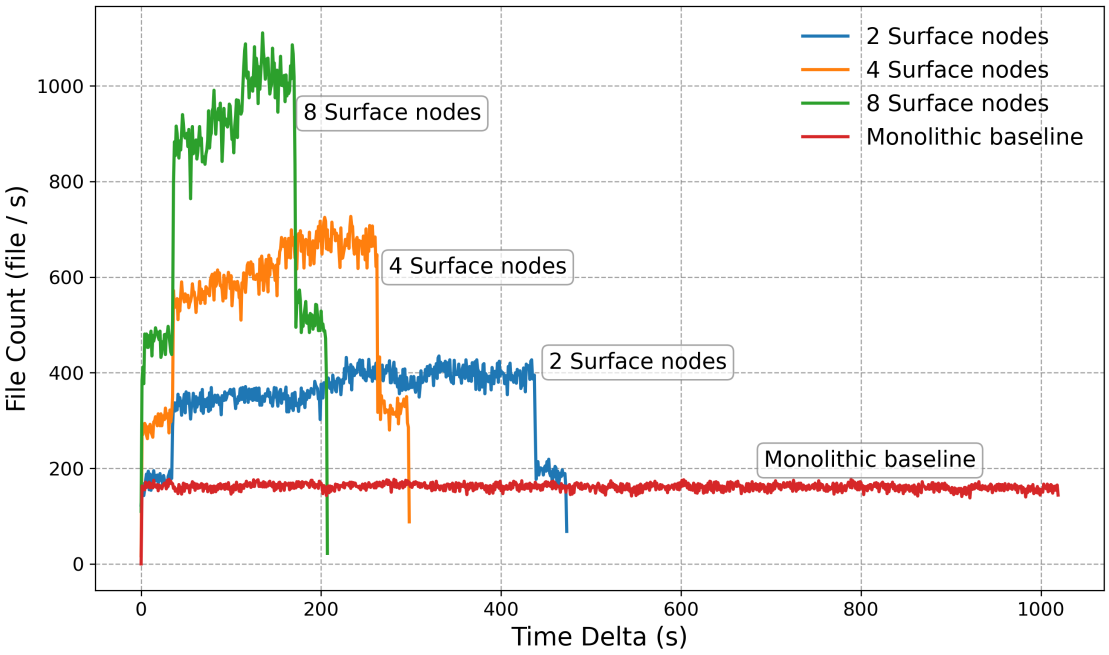


Figure 49: Ingestion rate over time with different numbers of surface nodes

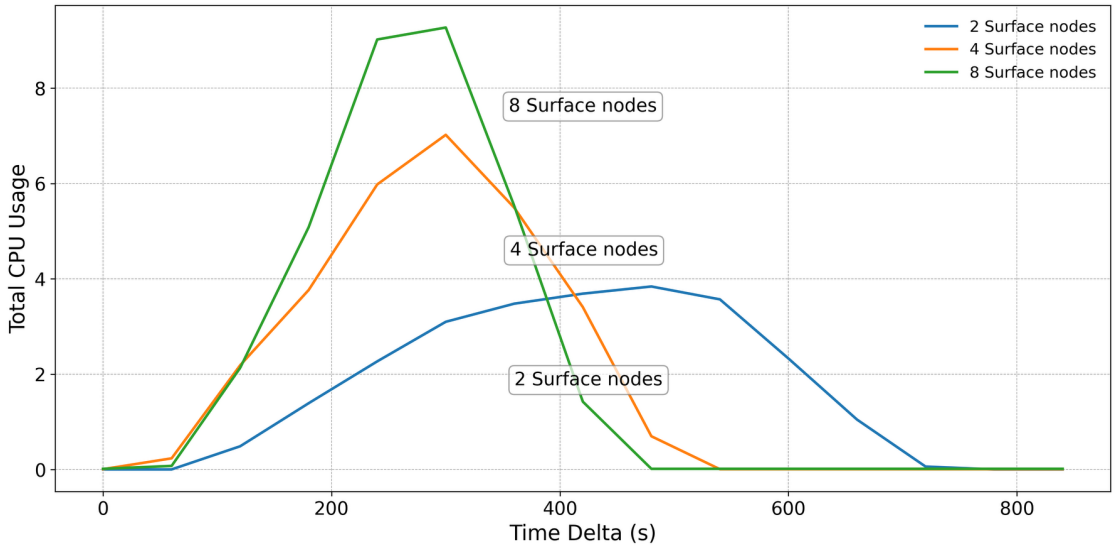


Figure 50: CPU usage over time with different numbers of surface nodes

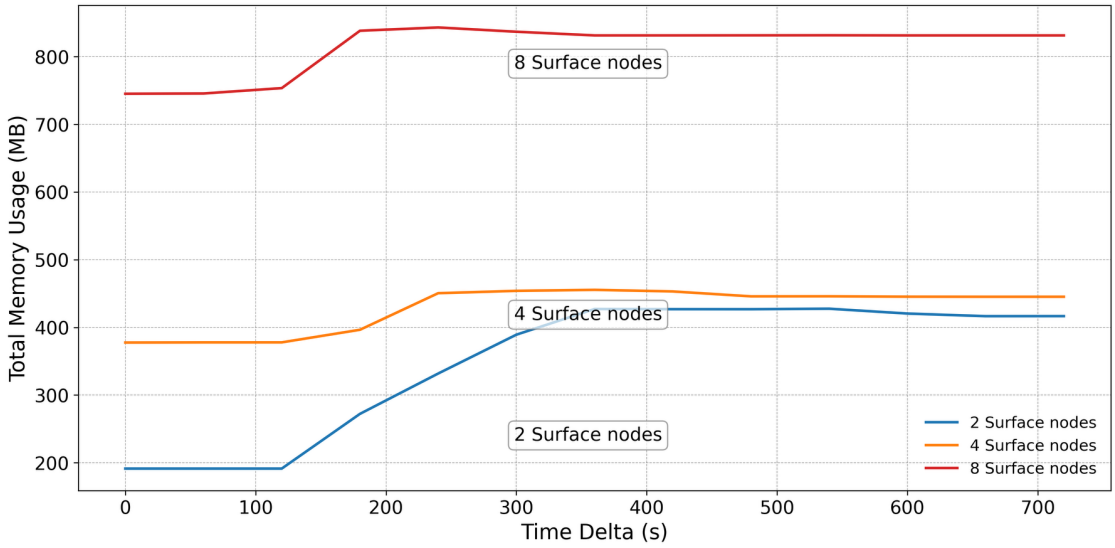


Figure 51: Memory usage over time with different numbers of surface nodes

13.4.3 Data ingestion

The purpose of the experiment is to demonstrate the prototype's ability to quickly ingest large volumes of data, and assess CPU and memory usage. We used a client to send as many ingestion requests as possible to the framework with dataset **L0** (refer to Table 12). The server setup included a Cassandra cluster with three nodes, an ElasticMQ cluster with one node, one ingestion scheduler, and two ingestion services. Three tests were performed with the same dataset, doubling the service instances

each time (two, four, and eight, respectively). Each test was repeated three times for consistency. Figure 49 shows data ingestion throughput in files per second. In each run, we scaled only the *Surface Service* instances, keeping the other configurations constant. Figure 50 and Figure 51 display CPU and memory usage, respectively. The results indicate that as we increased the number of service instances, the ingestion rate increased from 10 MB/s with two nodes to 35 MB/s with eight nodes. The time to ingest the entire dataset decreased accordingly. These results demonstrate that increasing service instances enables the system to handle more ingestion requests simultaneously. In addition, the resource footprint was minimal, optimizing resource usage.

In addition to our system's file ingestion rates, Figure 49 shows file ingestion rates to the monolithic pipeline. The monolithic pipeline peaks at 70 files per second, compared to 400 files per second for the *lakehouse architecture* at the lowest scaling configuration. Additionally, ingesting and processing the entire dataset takes approximately seconds with the monolithic approach versus just under 600 seconds with the *lakehouse architecture*. To ingest and construct the same knowledge graph, the *lakehouse architecture* requires roughly 650 seconds in total, while the monolithic pipeline, despite pre-constructing the knowledge graph, still requires at least seconds, excluding retrieval time.

13.4.4 Knowledge graph construction

One of our objectives in the proposed architecture is to create KGs on demand, reducing computation efforts and complexity by avoiding one large KG for the entire dataset. This approach optimally handles large data volumes and generates KGs as needed. We conducted experiments to demonstrate the prototype's ability to construct KGs on demand in reasonable time frames. We defined KGs of various sizes to represent different slice-and-dice operations and tested them on five datasets of varying sizes (see Table 12). Table 13 outlines the characteristics of each KG, including the expected number of statements and contexts. The online appendix [65] provides more details on KGs design. Dataset **L1** is small, resulting in short run times. Larger datasets **L2** and **L3** showed increased run times, while further expansion to **L4** stabilized run times. We use Apache Cassandra [71] for indexing—a distributed NoSQL database designed for managing large-scale datasets across distributed environments. Its architecture natively supports horizontal scalability and fault tolerance. In our experiments, we observed that as the index size grows, query response times remain stable and, in some cases, even improve due to optimized partitioning and caching mechanism. Note that **L1** is small and unrealistic for benchmarking. Realistic response times are seen in larger datasets, given the standard HTTP request timeout of 300 seconds. Figure 52 compares the response times for different KG constructions against datasets **L1**, **L2**, **L3**, **L4**, and **L5** (see Table 12 for more details).

To ensure semantic correctness, the OLAP operations applied in our system follow the formal KG-OLAP model, which defines safe aggregation over hierarchical dimensions. These semantics and their implementation are described in more detail in the supplemental material [67].

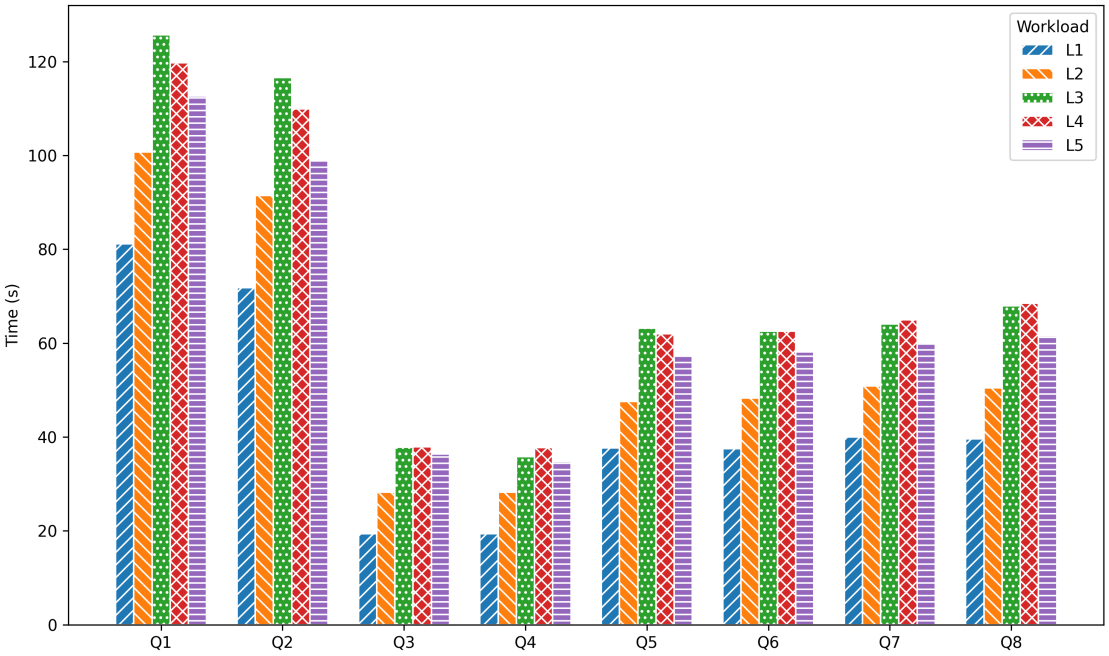


Figure 52: Run times of on-demand construction of different CKGs, specified by Q1–Q8, with various workloads (L1–L5), averaged over three runs

KG	Contexts	Quads
Q1	6 975	269 700
Q2	225	197 025
Q3	2 790	107 880
Q4	1	50 220
Q5	4 392	177 144
Q6	1	83 448
Q7	4 392	166 164
Q8	36	128 304

Table 13: Result size of the evaluation of the KG specifications (“queries”) in terms of returned contexts and quads

13.5 Conclusion

The increasing use of knowledge graphs (KGs) as a data source, coupled with the high volume, velocity, and variety of data required for their construction, highlights the limitations of monolithic architectures in supporting efficient analytical applications. In this paper, we proposed a cloud-native data lakehouse architecture for KG management, designed to address these challenges by enabling efficient data ingestion, indexing, and on-demand generation of contextualized KGs tailored to specific tasks.

We presented a proof-of-concept prototype using open-source frameworks and evaluated its performance using realistic synthetic data from the air traffic management domain. The results demonstrated that the architecture efficiently handles large-scale data ingestion and adapts dynamically to different load conditions. Ingestion rates increased near-linearly with the number of service instances, reaching up to 400 files per second with eight nodes, clearly outperforming the 70 files per second achieved by a monolithic pipeline. KG construction response times remained stable and reasonable, even as dataset sizes increased, demonstrating the system's scalability and reliability. Additionally, we compared our approach with a monolithic pipeline for constructing the same knowledge graph. The results clearly showed the superior performance, scalability, and resource efficiency of the proposed architecture, making it well-suited for high-velocity, data-rich environments.

This paper focused on efficient data ingestion and the on-demand generation of contextualized KGs for individual tasks. As future work, we plan to extend our approach with scalable graph operations over KG-OLAP cubes—such as abstraction, pivoting, and reification —to enable the transformation and aggregation of subgraphs within individual cube cells. These operations will support advanced analytics tasks, including link prediction, KG completion, and context-aware summarization.

We provide an online appendix [67] with detailed documentation of the KG Lakehouse system, including its architecture, microservice design, deployment plan, and a brief explanation of KG-OLAP. A companion repository [65] documents the experimental setup, infrastructure, datasets, benchmarking procedures, and performance results, along with evaluation logs and Jupyter notebooks for reproduction of experiments and analyses. The AIXM-based dataset generator [64] used to create the datasets used in the experiments is openly available, enabling reproduction. Finally, the source code of the prototype implementation [66], including deployment scripts and service-level configurations, is also provided.

14 References

[1] M. Schwarz, “Using a native RDF-based reasoning engine for the representation of contextualized knowledge graphs for air traffic management in the time-critical setting of the AWARE project,” masterthesis, Linz, Austria, 2025.

[2] G. Wakolbinger, “Using a relational database for the management of contextualized knowledge graphs for air traffic management in the time-critical setting of the AWARE project,” masterthesis, Linz, Austria, 2025.

[3] “ACHIEVING HUMAN-MACHINE COLLABORATION WITH ARTIFICIAL SITUATIONAL AWARENESS AWARE”, project fact sheet, HORIZON Europe. CORDIS, European Commission. Accessed: Apr. 16, 2025. [Online]. Available: https://cordis.europa.eu/project/id/101167442

[4] M. Schrefl, B. Neumayr, S. Gruber, M. Hartmann, I. Tukarić, and T. Radišić, “Creating an ATC knowledge graph in support of the artificial situational awareness system,” *Transportation Research Procedia*, vol. 64, 2022, pp. 328–336.

[5] A. Hogan *et al.*, “Knowledge Graphs,” *ACM Comput. Surv.*, vol. 54, no. 4, pp. 71:1–71:37, Jul. 2021, doi: 10.1145/3447772.

[6] R. Angles and C. Gutierrez, “Survey of graph database models,” *ACM Comput. Surv.*, vol. 40, no. 1, pp. 1:1–1:39, Feb. 2008, doi: 10.1145/1322432.1322433.

[7] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, “Foundations of Modern Query Languages for Graph Databases,” *ACM Comput. Surv.*, vol. 50, no. 5, pp. 68:1–68:40, Sep. 2017, doi: 10.1145/3104031.

[8] S. Cox, C. Little, J. R. Hobbs, and F. Pan, “Time Ontology in OWL.” Nov. 2022. Accessed: Jan. 07, 2025. [Online]. Available: https://www.w3.org/TR/owl-time/

[9] C. Gutierrez, C. Hurtado, and A. Vaisman, “Introducing time into RDF,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 19, pp. 207–218, Mar. 2007, doi: 10.1109/TKDE.2007.34.

[10] A. Rula, M. Palmonari, A. Harth, S. Stadtmüller, and A. Maurino, “On the Diversity and Availability of Temporal Information in Linked Open Data,” in *The Semantic Web – ISWC 2012*, P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. X. Parreira, J. Hendler, G. Schreiber, A. Bernstein, and E. Blomqvist, Eds., Berlin, Heidelberg: Springer, 2012, pp. 492–507. doi: 10.1007/978-3-642-35176-1_31.

[11] A. Rula *et al.*, “TISCO: Temporal Scoping of Facts,” in *Companion Proceedings of The 2019 World Wide Web Conference*, San Francisco USA: ACM, May 2019, pp. 959–960. doi: 10.1145/3308560.3316524.

[12] A. Piscopo, L.-A. Kaffee, C. Phethean, and E. Simperl, “Provenance Information in a Collaborative Knowledge Graph: An Evaluation of Wikidata External References,” in *The Semantic Web – ISWC 2017: 16th International Semantic Web Conference, Vienna, Austria, October 21–25, 2017,*

Proceedings, Part I, Berlin, Heidelberg: Springer-Verlag, Oct. 2017, pp. 542–558. doi: 10.1007/978-3-319-68288-4_32.

[13] K. Belhajjame *et al.*, "PROV Model Primer." Apr. 2013. Accessed: Dec. 18, 2024. [Online]. Available: https://www.w3.org/TR/prov-primer/

[14] P. A. Bonatti, A. Hogan, A. Polleres, and L. Sauro, "Robust and scalable Linked Data reasoning incorporating provenance and trust annotations," *Journal of Web Semantics*, vol. 9, no. 2, pp. 165–201, Jul. 2011, doi: 10.1016/j.websem.2011.06.003.

[15] J. McCarthy, "Notes on formalizing context," in *Proceedings of the 13th international joint conference on Artifical intelligence - Volume 1*, in IJCAI'93. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Aug. 1993, pp. 555–560.

[16] R. Guha, R. McCool, and R. Fikes, "Contexts for the semantic web," in *Proceedings of the 3rd International Conference on Semantic Web Conference*, in LNCS-ISWC'04. Berlin, Heidelberg: Springer-Verlag, Nov. 2004, pp. 32–46. doi: 10.1007/978-3-540-30475-3_4.

[17] R. Cyganiak, D. Wood, and M. Lanthaler, "RDF 1.1 Concepts and Abstract Syntax." Feb. 2014. Accessed: Jan. 07, 2025. [Online]. Available: https://www.w3.org/TR/rdf11-concepts/

[18] V. Nguyen, O. Bodenreider, and A. Sheth, "Don't like RDF reification? Making statements about statements using singleton property," in *Proceedings of the 23rd international conference on World wide web*, in WWW '14. New York, NY, USA: Association for Computing Machinery, Apr. 2014, pp. 759–770. doi: 10.1145/2566486.2567973.

[19] B. Neumayr, "Proof-of-concept KG system (AISA Deliverable D4.1)," Technical Report, Feb. 2021. Accessed: Oct. 31, 2024. [Online]. Available on the project's CORDIS page: https://doi.org/10.3030/892618

[20] R. Dividino, S. Sizov, S. Staab, and B. Schueler, "Querying for provenance, trust, uncertainty and other meta knowledge in RDF," *Journal of Web Semantics*, vol. 7, no. 3, pp. 204–219, Sep. 2009, doi: 10.1016/j.websem.2009.07.004.

[21] O. Udrea, D. R. Recupero, and V. S. Subrahmanian, "Annotated RDF," *ACM Trans. Comput. Logic*, vol. 11, no. 2, pp. 10:1–10:41, Jan. 2010, doi: 10.1145/1656242.1656245.

[22] A. Zimmermann, N. Lopes, A. Polleres, and U. Straccia, "A General Framework for Representing, Reasoning and Querying with Annotated Semantic Web Data." arXiv, Mar. 2011. doi: 10.48550/arXiv.1103.1255.

[23] L. Serafini and M. Homola, "Contextualized knowledge repositories for the Semantic Web," *Journal of Web Semantics*, vol. 12–13, pp. 64–87, Apr. 2012, doi: 10.1016/j.websem.2011.12.003.

[24] C. G. Schuetz, B. Neumayr, M. Schrefl, E. Gringinger, A. Vennesland, and S. Wilson, "The Case for Contextualized Knowledge Graphs in Air Traffic Management," in CEUR workshop proceedings, vol. CKG 2018. Monterey, CA, U.S.A., 2018.

[25] C. G. Schuetz, L. Bozzato, B. Neumayr, M. Schrefl, and L. Serafini, “Knowledge Graph OLAP: A multidimensional model and query operations for contextualized knowledge graphs,” *Semantic Web*, vol. 12, no. 4, pp. 649–683, Jun. 2021, doi: 10.3233/SW-200419.

[26] D. Brickley and R. V. Guha, Eds., “RDF Schema 1.1.” Accessed: Mar. 15, 2025. [Online]. Available: https://www.w3.org/TR/rdf-schema/

[27] J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler, “Named graphs, provenance and trust,” in *Proceedings of the 14th international conference on World Wide Web*, in WWW ’05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 613–622. doi: 10.1145/1060745.1060835.

[28] D. Beckett, T. Berners-Lee, E. Prud’hommenaux, and G. Carothers, “RDF 1.1 Turtle.” Accessed: Feb. 17, 2025. [Online]. Available: https://www.w3.org/TR/turtle/

[29] S. Harris and A. Seaborne, Eds., “SPARQL 1.1 Query Language.” Accessed: Feb. 17, 2025. [Online]. Available: https://www.w3.org/TR/sparql11-query/

[30] S. Sakr and G. Al-Naymat, “Relational processing of RDF queries: A survey,” *SIGMOD Rec.*, vol. 38, no. 4, pp. 23–28, Jun. 2010, doi: 10.1145/1815948.1815953.

[31] Y. Luo, F. Picalausa, G. H. L. Fletcher, J. Hidders, and S. Vansummeren, “Storing and Indexing Massive RDF Datasets,” in *Semantic Search over the Web*, R. De Virgilio, F. Guerra, and Y. Velegrakis, Eds., Berlin, Heidelberg: Springer, 2012, pp. 31–60. doi: 10.1007/978-3-642-25008-8_2.

[32] Z. Ma, M. A. M. Capretz, and L. Yan, “Storing massive Resource Description Framework (RDF) data: A survey,” *The Knowledge Engineering Review*, vol. 31, no. 4, pp. 391–413, Sep. 2016, doi: 10.1017/S0269888916000217.

[33] J. Broekstra, A. Kampman, and F. van Harmelen, “Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema,” in *The Semantic Web — ISWC 2002*, I. Horrocks and J. Hendler, Eds., Berlin, Heidelberg: Springer, 2002, pp. 54–68. doi: 10.1007/3-540-48005-6_7.

[34] C. Weiss, P. Karras, and A. Bernstein, “Hexastore: Sextuple indexing for semantic web data management,” *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 1008–1019, Aug. 2008, doi: 10.14778/1453856.1453965.

[35] J. J. Levandoski and M. F. Mokbel, “RDF Data-Centric Storage,” in *2009 IEEE International Conference on Web Services*, Jul. 2009, pp. 911–918. doi: 10.1109/ICWS.2009.49.

[36] M. Rodríguez-Muro and M. Rezk, “Efficient SPARQL-to-SQL with R2RML mappings,” *Journal of Web Semantics*, vol. 33, pp. 141–169, Aug. 2015, doi: 10.1016/j.websem.2015.03.001.

[37] S. Das, S. Sundara, and R. Cyganiak, “R2RML: RDB to RDF Mapping Language.” Accessed: Dec. 17, 2024. [Online]. Available: https://www.w3.org/TR/r2rml/

[38] M. Arenas, A. Bertails, E. Prud’hommeaux, and J. Sequeda, “A Direct Mapping of Relational Data to RDF.” Accessed: Jan. 08, 2025. [Online]. Available: https://www.w3.org/TR/rdb-direct-mapping/

[39] S. Kounev, K.-D. Lange, and J. Von Kistowski, *Systems Benchmarking*, vol. 1. Springer, 2020.

[40] D. J. Lilja, *Measuring Computer Performance: A Practitioner's Guide*. Cambridge: Cambridge University Press, 2000. doi: 10.1017/CBO9780511612398.

[41] F. Steiner, "Vergleich verschiedener Ansätze zur Umsetzung von Regeln in ATM Knowledge Graphs," (engl.: "*Comparison of different approaches for the implementation of rules in ATM Knowledge Graphs"*) Bachelor's thesis, Institute of Business Informatic – Data & Knowledge Engineering, Johannes Kepler University Linz, 2023.

[42] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *IEEE Trans. Knowl. Data Eng.*, vol. 1, no. 1, pp. 146–166, Mar. 1989, doi: 10.1109/69.43410.

[43] "5.4. Generated columns. PostgreSQL documentation." Accessed: Mar. 27, 2025. [Online]. Available: https://www.postgresql.org/docs/17/ddl-generated-columns.html

[44] M. J. Burzańska, P. Wiśniewski, and K. Stencel, "Recursive queries: Twenty-five years after SQL:1999," in *2024 IEEE international conference on big data (BigData)*, Dec. 2024, pp. 8272–8279. doi: 10.1109/BigData62323.2024.10825088.

[45] "7.8. WITH queries (common table expressions). PostgreSQL documentation." Accessed: Apr. 14, 2025. [Online]. Available: https://www.postgresql.org/docs/17/queries-with.html

[46] R. Kimball and M. Ross, The data warehouse toolkit: The definitive guide to dimensional modeling, 3rd ed. Wiley Publishing, 2013.

[47] M. Benerecetti, P. Bouquet, and C. Ghidini, "On the dimensions of context dependence," University of Trento, Departmental Technical Report, Nov. 2002. Accessed: Apr. 03, 2025. [Online]. Available: http://eprints.biblio.unitn.it/309/

[48] M. Homola, A. Tamilin, and L. Serafini, "Modeling contextualized knowledge," 2010. Accessed: Apr. 03, 2025. [Online]. Available: https://cris.fbk.eu/handle/11582/23789#

[49] I. D. Group, "Releases · sraoss/pg_ivm. GitHub." Accessed: May 20, 2025. [Online]. Available: https://github.com/sraoss/pg_ivm/releases

[50] R. Peters, Ed., "Cron," in *Expert Shell Scripting*, Berkeley, CA: Apress, 2009, pp. 81–85. doi: 10.1007/978-1-4302-1842-5_12.

[51] "19.5. Write Ahead Log," *PostgreSQL Documentation*. Feb. 2025. Accessed: Mar. 03, 2025. [Online]. Available: https://www.postgresql.org/docs/17/runtime-config-wal.html

[52] "CREATE TABLE. PostgreSQL documentation." Accessed: Apr. 17, 2025. [Online]. Available: https://www.postgresql.org/docs/17/sql-createtable.html

[53] "14.3. Controlling the Planner with Explicit JOIN Clauses," *PostgreSQL Documentation*. Feb. 2025. Accessed: Feb. 20, 2025. [Online]. Available: https://www.postgresql.org/docs/17/explicit-joins.html

[54] AISA: AI Situational Awareness Foundation for Advancing Automation. https://doi.org/10.3030/892618

[55] AWARE: Achieving Human–Machine Collaboration with Artificial Situational Awareness. https://doi.org/10.3030/101167442

[56] AIXM: Aeronautical information exchange model. https://www.aixm.aero (2019)

[57] Ali, W., Saleem, M., Yao, B., Hogan, A., Ngomo, A.C.N.: A survey of RDF stores & SPARQL engines for querying knowledge graphs. The VLDB Journal **31**(3), 603–628 (2021). https://doi.org/10.1007/s00778-021-00711-3

[58] Amazon: Neptune, https://aws.amazon.com/neptune/, accessed: 2025-05-13

[59] Apache Hadop: Hadoop. https://hadoop.apache.org/, accessed: 2025-12-01

[60] Apache Jena: Fuseki, https://jena.apache.org/documentation/fuseki2/, accessed: 2024-12-16

[61] Apache Spark: GraphX, https://spark.apache.org/graphx/, accessed: 2025-05-13

[62] Armbrust, M., Ghodsi, A., Xin, R., Zaharia, M.: The data lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. Communications of the ACM **64**(9), 52–59 (2021)

[63] Armbrust, M., Zaharia, M., Ghodsi, A., Xin, R.: Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In: 11th Conference on Innovative Data Systems Research, CIDR 2021, http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[64] AIXM data generator. https://doi.org/10.5281/zenodo.15767847

[65] KG Lakehouse experiments. https://doi.org/10.5281/zenodo.15767856

[66] KG Lakehouse demonstrator. https://doi.org/10.5281/zenodo.15767850

[67] KG Lakehouse documentation. https://doi.org/10.5281/zenodo.15767854

[68] Barry, M., Bifet, A., Chiky, R., El Jaouhari, S., Montiel, J., El Ouafi, A., Guerizec, E.: Stream2graph: Dynamic knowledge graph for online learning applied in large- scale network. In: 2022 IEEE International Conference on Big Data (Big Data), pp. 2190–2197 (2022). https://doi.org/10.1109/BigData55660.2022.10020885

[69] Beheshti, A., Benatallah, B., Motahari Nezhad, H.R.: Scalable graph-based olap analytics over process execution data. Distributed and Parallel Databases **34** (2015). https://doi.org/10.1007/s10619-014-7171-9

[70] Blazegraph: Blazegraph Database, https://blazegraph.com/, accessed: 2025-05-13

[71] Cassandra, A.: Cassandra. https://cassandra.apache.org, accessed: 2023-02-01

[72] Celina, V., Samardžić, K., Tukarić, I., Radišić, T., Hermann, R.H.: Gaze analysis of air traffic contoller using AI-based conflict detection. In: 2024 AIAA DATC/IEEE 43rd Digital Avionics Systems Conference (DASC) (2024). https://doi.org/10.1109/DASC62030.2024.10749617

[73] Clickhouse: Clickhouse. https://clickhouse.com/, accessed: 2024-07-01

[74] Databricks: The databricks lakehouse platform, https://www.databricks.com/ product/data-lakehouse, accessed: 2023-02-14

[75] Diefenbach, D., Wilde, M.D., Alipio, S.: Wikibase as an infrastructure for knowl- edge graphs: The eu knowledge graph. In: Hotho, A., Blomqvist, E., Dietze, S., Fokoue, A., Ding, Y., Barnaghi, P., Haller, A., Dragoni, M., Alani, H. (eds.) ISWC 2021. pp. 631–647. Springer (2021). https://doi.org/10.1007/978-3-030-88361-4_37

[76] ElasticMQ: Elasticmq. https://github.com/softwaremill/elasticmq, accessed: 2025-02-01

[77] Gassauer-Fleissner, S., Shabat, Z.B.: Financial crime discovery using amazon eks and graph databases. Tech. rep. (2022), https://aws.amazon.com/blogs/ architecture/financial-crime-discovery-using-amazon-eks-and-graph-databases/

[78] Georgiou, H., Pelekis, N., Sideridis, S., Scarlatti, D., Theodoridis, Y.: Semantic- aware aircraft trajectory prediction using flight plans. International Journal of Data Science and Analytics **9**(2), 215–228 (2020)

[79] gRPC: gRPC. https://grpc.io/, accessed: 2025-12-01

[80] Harby, A.A., Zulkernine, F.: From data warehouse to lakehouse: A comparative review. In: 2022 IEEE International Conference on Big Data. pp. 389–395 (2022). https://doi.org/10.1109/BigData55660.2022.10020719

[81] Hogan, A., Blomqvist, E., Cochez, M., D'amato, C., Melo, G.D., Gutierrez, C., Kirrane, S., Gayo, J.E.L., Navigli, R., Neumaier, S., Ngomo, A.C.N., Polleres, A., Rashid, S.M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., Zimmermann, A.: Knowledge graphs. ACM Computing Surveys **54**(4) (2021). https://doi.org/ 10.1145/3447772

[82] Janev, V., Graux, D., Jabeen, H., Sallinger, E.: Knowledge Graphs and Big Data Processing (01 2020). https://doi.org/10.1007/978-3-030-53199-7

[83] Keller, R.M.: Building a knowledge graph for the air traffic management community. In: Companion Proceedings of the 2019 World Wide Web Conference. pp. 700–704 (2019). https://doi.org/10.1145/3308560.3317706

[84] Kona, P., Wallace, A.: Using a knowledge graph to power a semantic data layer for databricks, https://www.databricks.com/blog/2022/06/17/ using-a-knowledge-graph-to-power-a-semantic-data-layer-for-databricks.html, accessed: 2023-02-14

[85] Kubernetes: Kubernetes. https://kubernetes.io, accessed: 2023-02-01

[86] Laney, D.: 3D data management: Controlling data volume, velocity and variety. META Group Research Note **6** (2001)

[87] Li, X., Zeng, W., Wang, Z., Zhu, D., Xu, J., Yu, W., Zhou, J.: Graphar: An efficient storage scheme for graph data in data lakes (2024). https://doi.org/10. 48550/arXiv.2312.09577, https://arxiv.org/abs/2312.09577

[88] MinIO: Minio – multi-cloud object storage. https://min.io/, accessed: 2025-02-01

[89] Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: RDFox: A highly- scalable RDF store. In: Arenas, M., Corcho, Ó., Simperl, E., Strohmaier, M., d'Aquin, M., Srinivas, K., Groth, P., Dumontier, M., Heflin, J., Thirunarayan, K., Staab, S. (eds.) ISWC 2015 – Part II. LNCS, vol. 9367, pp. 3–20. Springer (2015). https://doi.org/10.1007/978-3-319-25010-6_1

[90] Neo4j: Neo4j, https://neo4j.com/, accessed: 2025-05-13

[91] Ontotext: GraphDB, https://graphdb.ontotext.com/, accessed: 2025-05-13

[92] Oxford Semantic Technologies: RDFox, https://www.oxfordsemantic.tech/rdfox, accessed: 2025-05-13

[93] Papadaki, M.E., Tzitzikas, Y., Mountantonakis, M.: A brief survey of methods for analytics over rdf knowledge graphs. Analytics **2**(1), 55–74 (2023). https://doi.org/ 10.3390/analytics2010004

[94] Pradhan, A., Todi, K.K., Selvarasu, A., Sanyal, A.: Knowledge graph genera- tion with deep active learning. In: 2020 International Joint Conference on Neu- ral Networks (IJCNN). pp. 1–8 (2020). https://doi.org/10.1109/IJCNN48605.2020. 9207515

[95] Purohit, S., Van, N., Chin, G.: Semantic property graph for scalable knowledge graph analytics. In: 2021 IEEE International Conference on Big Data (Big Data), pp. 2672–2677 (2021). https://doi.org/10.1109/BigData52589.2021.9671547

[96] Redis: Redis. https://redis.io/, accessed: 2025-12-01

[97] Schuetz, C.G., Neumayr, B., Schrefl, M., Gringinger, E., Wilson, S.: Semantics- based summarisation of atm information: Managing information overload in pilot briefings using semantic data containers. The Aeronautical Journal **123**(1268), 1639–1665 (2019). https://doi.org/10.1017/aer.2019.74

[98] Schuetz, C.G., Bozzato, L., Neumayr, B., Schrefl, M., Serafini, L.: Knowledge Graph OLAP: A multidimensional model and query operations for contextual- ized knowledge graphs. Semantic Web **12**(4), 649–683 (2021). https://doi.org/10. 3233/SW-200419

[99] Schuetz, C.G., Neumayr, B., Schrefl, M., Gringinger, E., Vennesland, A., Wilson, S.: The case for contextualized knowledge graphs in air traffic management. In: Ca- padisli, S., Cotton, F., Giménez-García, J.M., Haller, A., Kalampokis, E., Nguyen, V., Sheth, A.P., Troncy, R. (eds.) Joint Proceedings of the International Work- shops on Contextualized Knowledge Graphs, and Semantic Statistics co-located with 17th International Semantic Web Conference (ISWC 2018). CEUR Work- shop Proceedings, vol. 2317. CEUR-WS.org (2018), https://ceur-ws.org/Vol-2317/ article-10.pdf

[100] Sequeda, J., Lassila, O.: Designing and building enterprise knowledge graphs. Springer (2022)

[101] Serafini, L., Homola, M.: Contextualized knowledge repositories for the semantic web. Journal of Web Semantics **12-13**, 64–87 (2012). https://doi.org/10.1016/j. websem.2011.12.003, reasoning with context in the Semantic Web

[102] Shao, B., Wang, H., Li, Y.: Trinity: A distributed graph engine on a memory cloud, pp. 505–516 (06 2013). https://doi.org/10.1145/2463676.2467799

[103] Singh, A., Singh, V., Aggarwal, A., Aggarwal, S.: Event driven architecture for message streaming data driven microservices systems residing in distributed version control system. In: 2022 International Conference on Innovations in Science and Technology for Sustainable Development (ICISTSD). pp. 308–312 (2022). https://doi.org/10.1109/ICISTSD55159.2022.10010390

[104] Stardog: Stardog - the enterprise knowldge graph platform, https://www.stardog. com/, accessed: 2023-02-14

[105] Steiner, D., Kovacic, I., Burgstaller, F., Schrefl, M., Friesacher, T., Gringinger, E.: Semantic enrichment of dnotams to reduce information overload in pilot briefings. In: 2016 Integrated Communications Navigation and Surveillance (ICNS). pp. 6B2– 1–6B2–13 (2016). https://doi.org/10.1109/ICNSURV.2016.7486359

[106] Verborgh, R., Sande, M.V., Hartig, O., Meester, B.D., Vocht, L.D., Mannens, E., de Walle, R.V.: Querying datasets on the web with high availability. In: Proceedings of the 13th International Semantic Web Conference (ISWC 2014). LNCS, vol. 8796, pp. 180–196. Springer (2014). https://doi.org/10.1007/978-3-319-11964-9_12

[107] Wieringa, R.: Design science methodology for information systems and software engineering. Springer (2014)

[108] Wikimedia Deutschland: Wikibase. https://wikiba.se/, accessed: 15 March 2024

[109] Xiao, G., Kontchakov, R., Calvanese, D., Compatangelo, E., Motik, B.: Virtual knowledge graphs: An overview of systems and use cases. Data Intelligence **1**(3), 201–223 (2019). https://doi.org/10.1162/dint_a_00012

[110] Zou, X.: A survey on application of knowledge graph. Journal of Physics: Confer- ence Series **1487**, 012016 (03 2020). https://doi.org/10.1088/1742-6596/1487/1/012016

[111] H. Knublauch and D. Kontokostas, Eds., “Shapes Constraint Language (SHACL).” Accessed: Jun. 17, 2025. [Online]. Available: https://www.w3.org/TR/shacl/