Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Improving Graph-Based Reasoning: Automated Logic Propagation with Industrial Graph Processing Systems

Scientific Students' Association Report

Author:

Máté Baksa
Inez Anna Papp
Martin Ruszka

Advisor:

Attila Ficsor
Dr. Kristóf Marussy
Dr. Oszkár Semeráth

2023

Contents

Kivonat

A Kiberfizikai Rendszerek (CPS) az elmúlt évtizedekben jelentős figyelmet kaptak az ipar és a kutatók részéről. A CPS-ek intelligens eszközökből épülő elosztott rendszerek, melyek több érzékelőt is használva képesek a különböző szenzorokkal mért adatokat gyűjteni, és a mérések alapján következtetni és beavatkozni.

Azonban a nagy elosztott rendszerek kifejlesztése komoly kihívást jelentő feladat, ami nagy mennyiségű számítási és integrációs feladattal jár. A különböző forrásból származó mérési adatok gyakran inhomogén tudásbázist eredményeznek, amely potenciálisan hiányos vagy akár inkonzisztensek lehetnek. Hatékonyan következtetni egy ilyen rendszerben kihívást jelentő feladat. Automatizáció nélkül csak a fejlesztőre támaszkodhatunk, ami egy sok munkát jelentő feladat lehet, valamint hiba esetén hibás lépéseket vezethet be (ami érvényteleníheti az egész tudásbázis tartalmát), vagy elmulaszthatja a következtetési lépéseket (ami a tudásbázist kevésbé pontossá teheti). Összefoglalva a rendszert inkonzisztenssé vagy hiányossá teheti.

Ezen értekezés célja olyan technika nyújtása, amely képes automatikusan kinyerni következtetési szabályokat a tudásbázisok számára. Ezenkívül teljesítménytesztek végrehajtása gráf adatbázisok hatékonyságának értékelésére, amikor őket szabványosított teszteknek és munkaterheléseknek vetnek alá. Ezek a tesztek objektív mértékeket kívánnak nyújtani az adatbázis képességeiről a grafikus feladatok kezelésében, például lekérdezésben, átkutatásban és gráf adatok frissítésében. Ezeknek a teszteknek az eredményei segítik a tudásbázis fejlesztőket és felhasználókat abban, hogy tájékozott döntéseket hozzanak a gráf adatbázis rendszerek kiválasztásában és optimalizálásában a specifikus alkalmazások és felhasználási esetek számára.

A dolgozatunkban egy olyan módszert mutatunk be, amely lehetővé teszi a mérési adatok hatékony tárolását egy gráfadatbázisban. Ezenkívül automatizált folyamatot nyújtunk a következtetési szabályok származtatásához, hogy a tárolt adatok helyességét vizsgáljuk. Különböző adatbázis méretekre kiterjedő lekérdezési értékeléseket végezünk, és összehasonlító elemzést végzünk alternatív adatbázisrendszerekkel szemben. Végül megvizsgáljuk a megoldásunkat a Trainbenchmark nevű mérési környezet segítségével.

Az optimális következtetési szabályok megtalálása lehetővé teszi számunkra, hogy szisztematikusan generáljuk a következtetési szabályokat, amelyek felgyorsítják a rendszer fejlesztését, valamint javíthatjuk a következtetési képességét. Ez a megközelítés lehetővé teszi, hogy hatékonyabban működjön az adatbázis, elősegítve annak pontos és gyors adatfeldolgozását, miközben minimalizálja a hibalehetőségeket.

Abstract

Cyber-Physical Systems (CPS) have gained significant attention from industry and researchers in recent decades due to their promising potential across various domains. CPSs are distributed systems built of intelligent devices that can use multiple sensors and manage data flows and operations. Moreover, it continuously monitors physical entities integrated as part of critical infrastructures in a knowledge base.

However, the development of an extensive distributed system is a challenging task, as it generates an enormous amount of implementation and resource management challenges. This creates a large set of inhomogeneous knowledge, which may be incomplete or inconsistent. Efficiently reason in such a system can be a challenging task. Without any automation, we can only depend on the slower user, which may introduce invalid reasoning steps (which can invalidate the whole knowledge base) or miss reasoning steps (which can make the knowledge base less precise). In summary, this can make the system inconsistent or incomplete.

This thesis aims to provide a technique that can automatically derive propagation rules for knowledge bases. Furthermore, benchmarking is used to assess the performance and efficiency of graph databases by subjecting them to standardized tests and workloads. These benchmarks objectively measure a database's capabilities in handling various graph-related tasks, such as querying, traversing, and updating graph data. The results of these benchmarks help knowledge base developers and users make informed decisions about selecting and optimizing graph database systems for specific applications and use cases.

Our report proposes an interface to store measurement data in a graph database efficiently. Moreover, we provide an automated process to derive propagation rules to reason over the correctness of the stored data. Conduct query evaluations across diverse database scales and perform comparative analysis against alternative database systems. Finally, we evaluate our solution with the Trainbecnhmark modeling benchmark.

Finding the way to make the right propagations allows us to systematically generate propagation rules, making the system more efficient and decreasing the risk of invalidating it.

Chapter 1

Introduction

1.1 Context

In an age dominated by data-driven decision-making and complex networked structures, the field of graph-based reasoning has emerged as a critical component in solving a wide range of real-world problems. From social network analysis to recommendation systems, and from transportation optimization to bioinformatics, the ability to efficiently process and reason about graph data structures has become increasingly essential. Graph processing systems, the workhorses of graph-based reasoning, provide the foundation for performing these operations at scale. However, as the complexity and size of graph data continue to grow, there is an urgent need for enhancements in both the theoretical and practical aspects of graph processing.

1.2 Problem Statement

However, the development of an extensive distributed system is a challenging task, as it generates an enormous amount of implementation and resource management challenges. This creates a large set of inhomogeneous knowledge, which may be *incomplete* or *inconsistent*. Collection and efficient reasoning in such a system can be a challenging task, which requires the merging of information coming from multiple sources.

Existing description logic based technologies promise high reasoning capacities, but reportedly provide low performance in real-world scenarios [9]. On the other hand, graph-based knowledge bases claim high performance at the cost of manually defined reasoning rules. However, without any automated derivation of reasoning, users introduce invalid reasoning steps (which can invalidate the whole knowledge base) or miss reasoning steps (which can make the knowledge base less precise). In summary, this can make the system inconsistent or incomplete.

Finally, modeling environments are typically defined by domain-specific languages, which include a metamodel and some well-formedness constraints. Those concepts need to be mapped to the data storage technology, which can be also challenging and error-prone.

1.3 Objectives

The objective of this thesis is to provide a technique to map domain-specific modeling languages to knowledge-base technologies. This includes the automatic derivation of propagation rules for knowledge bases from constraints.

Furthermore, the report aims to provide a performance benchmark used to assess the performance and efficiency of graph databases by subjecting them to standardized tests and workloads.

1.4 Contributions

Our report proposes an interface to store domain specific models in a graph database efficiently. In our report, we provide the following contributions

- We provide a mapping technique of metamodel concepts to the TypeDB knowledge base schema concepts.
- Moreover, we provide an automated process to derive propagation rules to reason over the correctness of the stored data.
- Finally, we evaluate our solution with the Trainbecnhmark modeling benchmark and conduct query evaluations across diverse database scales, and perform comparative analysis against alternative database systems.

1.5 Added Value

Finding the way to make the right propagation allows us to systematically generate propagation rules, making the system more efficient and decreasing the risk of invalidating it.

1.6 Structure of the Report

The rest of report is structured as follows:

- Chapter 2 gives an overview of the background of this report.
- Chapter 3 provides the mapping technique from metamodeling concepts to knowledge-base schema elements.
- Chapter 4 provides an algorithm that maps well-formedness rules to propagation rules.
- Chapter 5 provides an evaluation of the approach.
- Chapter 6 provides a brief overview of the related work.
- Chapter 7 concludes the report.

The following tools used to aid the writing of the report:

- Google Translate
- Grammarly
- ChatGPT
- Overleaf spell checking

Chapter 2

Preliminaries

2.1 Graphs as Logic Structures

First and foremost, I consider it important to provide a general overview of graph databases. The definition of “graph” comes from the field of mathematics. A graph contains a collection of data that highlights the connections between the different data entities.

Throughout a simple illustrative case, let’s represent the relationships of a group of people (*Figure 2.1*). In this example, we can see different entities with different relations, such as "friend" and "married". Clearly, if Dave friends with Dan, then it should be shown backward as well, so that is why in two cases the backward arrow is shown with a dashed line. It is possible to discover who has connected with whom and the nature of their relationship. As a result, the born of graph databases is evident, due to the potent expressiveness that a graph structure can offer.

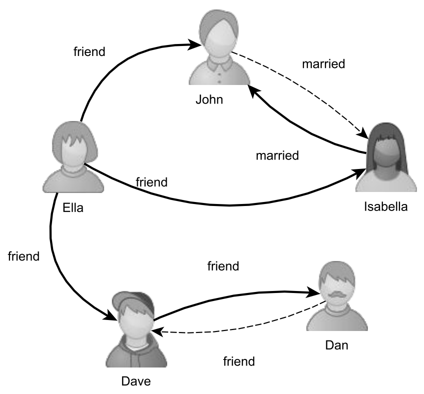


Figure 2.1: Simple illustration for a graph

Graph databases are categorized under the NoSQL data representation method. However, what sets them apart is that they connect data implicitly, where the relationships between data are of fundamental importance. Simply put, in the case of a graph database, the relationships between data are equally significant as the attributes itself. In contrast to relational databases, where we can establish relationships using JOIN operations with many tables. Over a large amount of data, our queries become complex once they surpass a certain level of complexity. This is where the graph database comes into play, as relationships are crucial not only in query execution but also in the structure of the database, resulting in simplifying our queries.

Entities refer to tables, and relationships to foreign keys in relational databases, while in graph representation tables can be compared to nodes and foreign keys as edges. You can build a complex database from this data structure. Throughout the thesis, we aimed to understand and implement this with TypeDB (*Section 2.2*).

A knowledge base[6] is an organized collection of information used to store, manage and retrieve data. It provides a foundation for storing and organizing data in a way that makes it accessible and meaningful. Due to inference, TypeDB (*Section 2.2*) is claiming to the title of the knowledge base. In the context of databases and knowledge bases, inference is the process of deriving new information or insights from existing data through logical reasoning or rules (*Section 2.2.4*).

2.2 TypeDB

The development by Vaticle is the TypeDB database a completely new and promising database paradigm. In TypeDB, a database is made up of two components: a schema and data. A schema refers to the structure or organization of the data within the database. TypeDB databases employ static typing with a strict type hierarchy that is maintained through inheritance. Everything that is not permitted by the schema is considered forbidden in the database. TypeDB is capable of reasoning over data using a set of user defined rules. The reasoning engine uses rules as a set of logic to infer new data, based on the existence of patterns in data of a TypeDB database.

TypeDB, consisting of entity, relationship, and attribute types, as well as type hierarchies, roles, and rules, enables higher-level thinking compared to traditional relational tables. All queries to a TypeDB database are written in TypeQL, which is a declarative query language. TypeQL serves as both a Data Definition Language (DDL) and a Data Manipulation Language (DML).

As they say, TypeDB has managed to create a database that is able to solve the lack of expressivity in standard relational[1] and document[2] databases. The database offers many possibilities, but let's start with the basics.

2.2.1 Running example

In this example that follows (*Figure 2.2*), we can see a running query in TypeDB, which was implemented in the Train Benchmark (*Section 2.3*). In brief, a TypeDB data structure and database involves defining a schema, inserting data, and executing queries as operations. Through this reference, we gain insight into creating a database, querying, and managing rules using the RouteSensor example. The purpose of the query is to identify a missing edge under certain conditions.

[1]e.g.:PostgerSQL[13]
[2]e.g.:MongoDB[3]

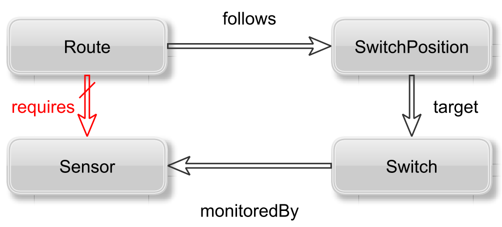


Figure 2.2: RouteSensor (*A running example in TypeDB*)

2.2.2 Schema

When you build a graph database in TypeDB, defining a schema is the primary step that needs to be taken. A schema is a collection of different types and rules that define the structure and constraints of data within a particular database or system.

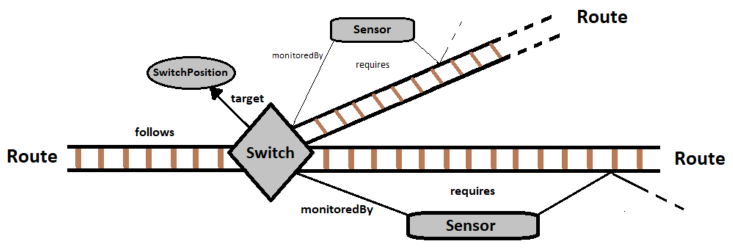


Figure 2.3: Diagram of RouteSensor

The basis of our data is a railway network model. The nodes of the model include regions, routes, sensors, semaphores, segments of a route, and switches. A node possesses certain characteristics, which are known as attributes. There is inheritance in TypeDB and we can define abstract entities, such as the base object RailwayContainer.

Example 1: The following is a part of the defined schema in TypeQL code.

```
define

    RailwayContainer sub entity, abstract, owns id; // Abstract class

    Route sub RailwayContainer, owns active, owns entry, owns exit,
    plays requires:Route, plays follows:Route; // Concrete class with references
    ...
    id sub attribute, value long; // Attribute
    ...
    requires sub relation, relates Route, relates Sensor; // Reference
    ...
```

To define a schema, it must start with the keyword "define". All listed nodes inherit the RailwayContainer properties (*Figure 2.5*). In the provided code, we can see that the "Railwaycontainer" entity is declared as abstract, and the "Route" entity inherits its "id" attribute. After defining the necessary nodes and their properties, the next step is to implement the connection between them.

In contrast to relational databases [1], where relationships between tables are established through foreign keys, TypeQL uses roles declared with the keyword "plays" to define entity connections. Both nodes and edges can have properties and play a role in another relationship. In the RouteSensor (*Figure 2.2*), no additional attributes are declared for the relations.

2.2.3 Query

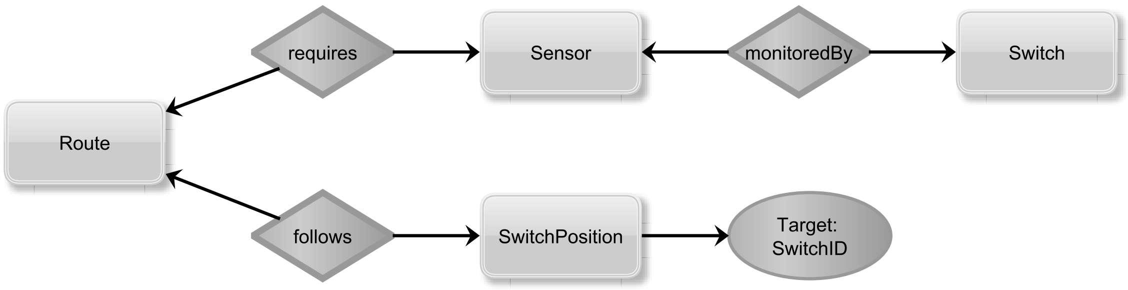


Figure 2.4: Relationships of RouteSensor Query

Sensors play a critical role in ensuring the safety of traffic on these routes, the query of RouteSensor (*Figure 2.2*) examines its absence. As illustrated in (*Figure 2.4*), there is a 'requires' connection between the Route and Sensor Entity, a 'monitoredBy' connection between Sensor and Switch, and a 'follows' relationship between Route and SwitchPosition entity. It's important to note that there is no direct connection between the Switch and SwitchPosition object, only instead, the SwitchPosition entity stores the "SwitchID" in its target attribute.

Example 2: The RouteSensor query in TypeQL language:

```
match
    $route isa Route, has id $routeID;
    $switchPosition isa SwitchPosition, has id $switchPositionID, has target $target;
    $sensor isa Sensor, has id $sensorID;
    $switch isa Switch, has id $switchID;
    (Route: $route, SwitchPosition: $switchPosition) isa follows;
    (TrackElement: $switch, Sensor: $sensor) isa monitoredBy;
    $switchID=$target;
    not {(Route: $route, Sensor: $sensor) isa requires;};
get
    $routeID, $sensorID, $switchPositionID, $switchID;
```

The "match" keyword specifies the model that the query should match in the knowledge graph. Firstly, starts by defining variables ("$route") for entities and their keys ("$routeID"). Instances of these entity types are represented by these variables in the remaining query. After, it specifies that there should be a "follows" relationship type between Route↔SwitchPosition and a "monitoredBy" relationship between TrackElement(Switch)↔Sensor. Furthermore, it states, that the "$target" property of the "$switchPosition" entity is equal to the "$switchID" of the "$switch" entity. There is a negation statement (*Figure 2.2*) in the query because we are searching for the absence of the "requires" edge between Route↔Sensor. Finally, with the keyword "get", we will receive the attributes, which are matching the previous conditions.

The purpose of this query is to continuously validate the model, and ensure that any sensors connected to a switch on a specific route must also have a direct connection to that same route. This query is intended to confirm that there are no circular connections, which will be important to the benchmark (*Section 2.3*) to assess navigation efficiency and identify negative conditions.

2.2.4 Rules

Cyber-Physical Systems (CPS)[17] is a new level of digital systems, composed of computational and physical capability. This requires new level of modeling tools to represent sensor data collected by a distributed system. That is why we need a database capable of storing every detail and connection that exists within their domain, both digital and physical, and applying logic and reason to help extend critical information from it.

TypeDB provides the ability to use logical reasoning. Rules define embedded logic as a part of the schema. Reasoning or inference, is performed at a query time. Inference from rules is only available for read transactions.

The infer option in TypeDB is related to the reasoning capabilities of the system. Inference or reasoning is one of the key features that sets Vaticle TypeDB apart from other database systems. Through inference, Vaticle can derive new knowledge from the data and the defined schema without the need for explicitly storing every piece of information.

2.2.4.1 Propagation rules

In order to demonstrate the operation and purpose of propagation rules, we first need to define error patterns. Error patterns describe the malformedness of a model.

Example 3: For example RouteSensor ((*Figure 2.2*)) can be seen as an error pattern.

```
match
    $route isa Route, has id $routeID;
    $switchPosition isa SwitchPosition, has id $switchPositionID, has target $target;
    $sensor isa Sensor, has id $sensorID;
    $switch isa Switch, has id $switchID;
    (Route: $route, SwitchPosition: $switchPosition) isa follows;
    (TrackElement: $switch, Sensor: $sensor) isa monitoredBy;
    $switchID=$target;
    not {(Route: $route, Sensor: $sensor) isa requires;};
get
    $routeID, $sensorID, $switchPositionID, $switchID;
```

The name of the error pattern is routeSensor, its parameters are a route, a sensor, a switchPosition and a switch. We can specify the type of a parameter like switch(sw), in that case sw is a switch. We can also look at that as a vertex of the graph. With writing for example follows(route, swP), we can specify that there is a connection between route and swP named follows. In the context of graphs this connection is a directed edge of the graph. When all constraints are met (sw is a switch, route is a route, etc.), the error pattern evaluates to true indicating the malformedness of the model.

If there is a malformedness in the model, we would like to avoid that. That is where the propagation rules help us. They can propagate when a model is only one step to satisfy an error pattern (there is one edge that needs to be indented or deleted, or there is one type that needs to be specified or specified not to be a specific type), the rule can propagate that.

Propagation rules also work when there is a predicate, that describes the welformedness of the model. In the following example the model can only satisfy all wellformedness constaints is it satisfies routeSensor to every route, sensor, switchPosition and switch.

2.3 The Train Benchmark

In the model-driven design of critical systems, such as in the automotive industry, aviation electronics, or train control systems, it is essential to identify potential errors in the system databases as early as possible. Since correcting design errors in later stages of development is significantly more costly. Therefore, it is inevitable to immediately detect violations of good modeling practices. This means introducing rule violations by engineers or some automated model manipulation steps immediately after making certain model modifications. Therefore, industrial design tools verify the correctness of a model by reviewing its constraints again after specific modifications have been made to the model.

The Train Benchmark[14] is a macro-benchmark spanning multiple technologies designed to measure the performance of continuous model validation with graph-based models and constraints specified as queries. First, it loads and validates an automatically generated (increasing in size) model, followed by a few transformations to the model then, the model is changed by some transformations, immediately followed by revalidation of the constraints.

The Train Benchmark was designed to comply with the four criteria defined in [5] for domain-specific benchmarks:

1. Relevance - to measure systems peak performance and price/performance, when evaluating typical operations on the problem domain.
2. Portability - easy to implement on many different systems
3. Scalability - should apply to small and large computer systems.
4. Simplicity - must be understandable, or else its credibility is compromised.

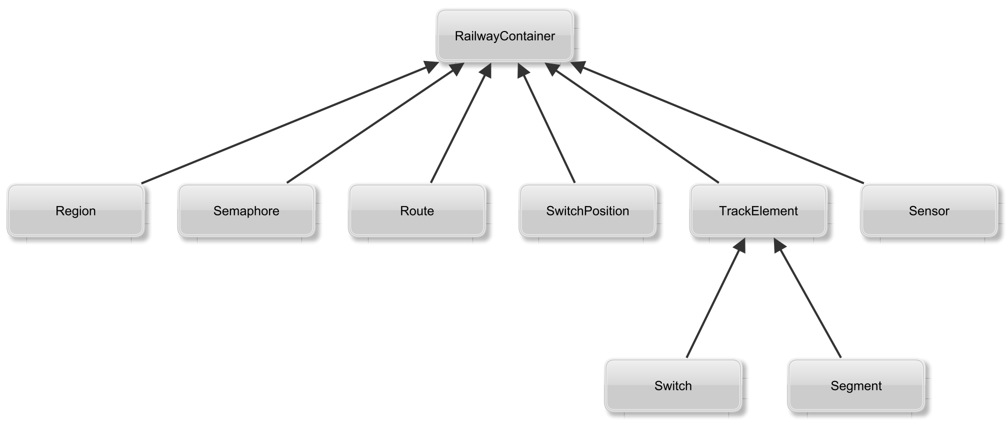


Figure 2.5: Entity Hierarchy

2.3.1 Structure of The Benchmark

The train benchmark data structure is a network railway model (*Figure 2.5*). The network is divided into different Regions. In a given region, there may be multiple Routes, each representing railway paths. Ensuring the safety of traffic on these routes is heavily reliant on the functioning of Sensors. They play a critical role in detecting and preventing potential accidents, making them an essential component of the overall safety system. The track elements (*Switches and Segements*) are in Route so they are also monitored by Sensors. A Route is linked to another Route through a SwitchPosition. Each Route is equipped with a Semaphore at its entry and exit points. The entity relationships of the model can be found on the Figure 2.6 representation.

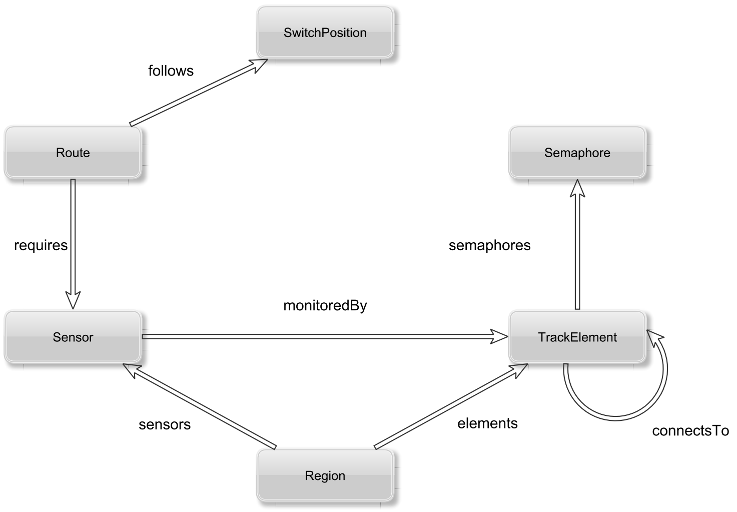


Figure 2.6: Entity Relationship

2.3.2 Data Generation for Benchmarking

In the Train Benchmark, a benchmark case configuration is defined by three primary inputs: a scenario, an instance model size, and a set of queries. The scenario sets the specific characteristics of the model, such as error percentages, while the transformation is based on both the scenario and a selected query.

To generate the instance models used in the Train Benchmark, the framework employs its generator module. This model generator relies on a pseudorandom number generator with a constant random seed to guarantee the reproducibility of results. Additionally, you have the ability to set the execution configuration to specify minimum and maximum heap memory. You can specify the minimum and maximum range of model sizes for benchmarking. As a result, models will be created within this range, with each model size being twice as large as the previous one. Furthermore, you can define the outcome for which models will be generated, these are the Batch, Inject, and Repair scenarios. Finally, you can choose the formats in which the models will be generated, for instance in TypeDB, SQL or in Neo4J [15], etc.

2.3.3 Queries for Benchmarking

The Train Benchmark has defined four benchmark phases for validating the model: read, check, transformation, and recheck. In the batch scenario, the instance model is loaded from storage in the read phase. During the check phase, model validation is performed by executing the queries. In the batch case, the benchmark uses a model free of errors. During the benchmark, we intentionally inject errors and then fix them. Therefore, we make the necessary repairs to correct any mistakes. Queries are used to capture and verify the well-formedness constraints.

In the graph representation of queries, rectangles represent entities, double-line arrows represent relationships, and ellipses represent entity attributes. The single arrows indicate that the value of one attribute should match the value of another attribute. When you see red colored arrows and attributes, it means that they are absent or they violate the model.

2.3.3.1 PosLength

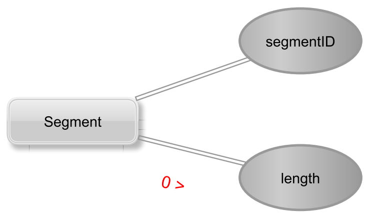


Figure 2.7: Graph of Postlength Query

The requirement states that a segment must have a length that is greater than zero. The associated query (*Figure 2.7*) defines a simple property check, which is a frequent use case in the context of validation.

2.3.3.2 SwitchMonitored

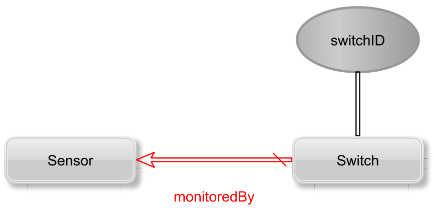


Figure 2.8: Graph of SwitchMonitored Query

It is required that each switch is connected to at least one sensor. The following query (*Figure 2.8*) is used to check if there is a connection between two vertices. This pattern is commonly used in more complex queries, such as the RouteSensor query.

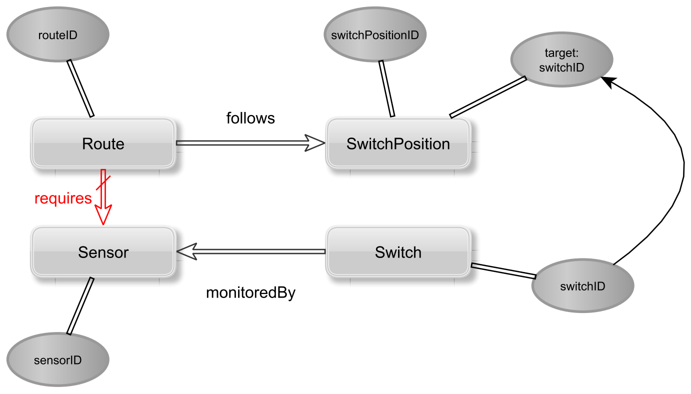


Figure 2.9: Graph of RouteSensor Query

2.3.3.3 RouteSensor

For a switch to belong to a route, all sensors associated with it must also be linked directly to the same route. This query (*Figure 2.9*) is described in more detail in the Section 2.2.3.

2.3.3.4 SwitchSet

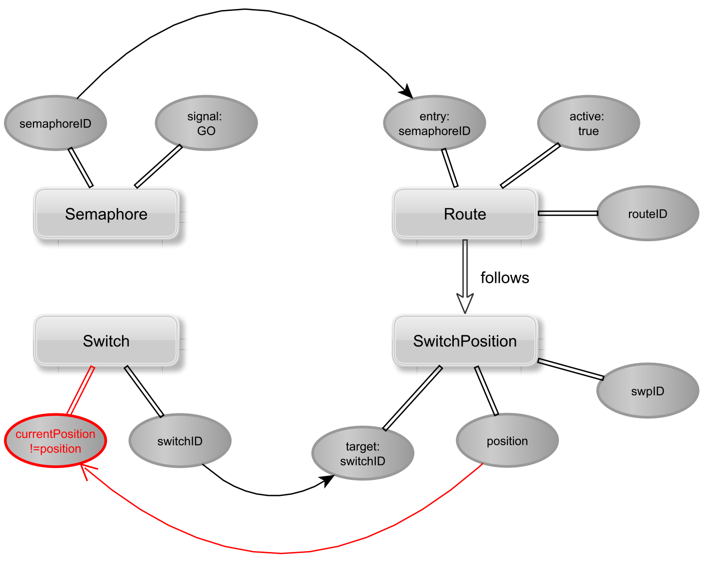


Figure 2.10: Graph of SwitchSet Query

It is necessary for the entry semaphore of an active route to have a "GO" signal only when all switches along the route are in the correct position as specified for the exact route. The related query (*Figure 2.10*) evaluates how well navigation and filtering operations perform.

2.3.4 Received Results of The Benchmark

After running a benchmark case successfully, we record the execution times for each phase and the number of invalid elements. Additionally, the result set must include the identifiers of the elements to allow for solution correctness checking by the framework.

Chapter 3

Mapping Domain-Specific Languages to Knowledge Base Concepts

The key objective of this section is to establish a comprehensive understanding of the procedures necessary for generating diverse sets of databases within TypeDB (*Section 2.2*). Additionally, it aims to facilitate benchmarking within the database by executing queries of varying complexities.

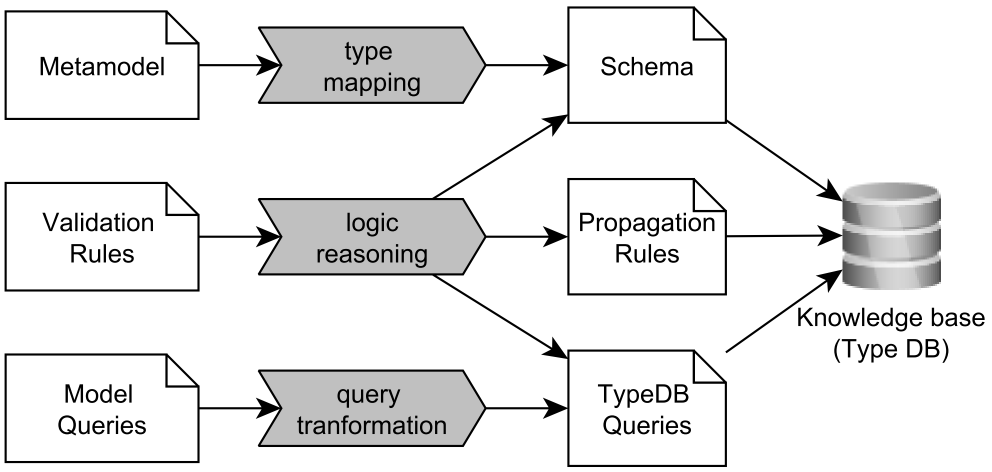


Figure 3.1: Functional overview of our approach

3.1 Transforming the metamodel into a TypeDB Schema

Before the initiation of the data insertion process into a TypeDB database, it is crucial to establish and articulate a well-defined schema (*Section 2.2.2*) that outlines the structure and organization of the data. Fortunately, TrainBenchmark (*Section 2.3*) streamlines this phase by already providing the pre-defined schema that requires implementation. This pre-existing schema serves as a foundational framework, expediting the implementation process by offering a structured outline for the database organization and data attributes before the actual insertion of data.

3.1.1 Classes of the metamodell

RailwayContainer(abstract): This serves as the foundational class from which all other classes in our railway system inherit. As an abstract class, the RailwayContainer establishes common attributes and methods which are vital for the diverse components of the railway network. By acting as the parent class, it ensures a consistent framework and structure for the subsequent child classes.

Region: The Region class plays a pivotal role in segmenting the railway network. It acts as a container for all the Routes, providing a structured way to manage and organize different paths within the system. Given its nature, it simplifies the understanding and representation of the railway's geography.

Semaphore: Essential for safe railway operations, the Semaphore class represents the railway signals. These signals regulate train traffic, ensuring that there are no collisions and that trains can navigate the network safely and efficiently.

Route: The Route class details a specific path or journey within the railway network. Every route is defined by its starting and ending points, which are semaphores. This encapsulation provides a clear depiction of how trains can navigate from one point to another within the system.

SwitchPosition: This class is integral to dynamic route management. It denotes the intended position of a Switch element, ensuring that a specific route is activated or deactivated based on operational needs. By controlling switches, this class allows for real-time adaptability in the railway network.

TrackElement(abstract): As an abstract class, TrackElement sets the stage for two specific elements in the railway system: Switch and Segment. It establishes common properties and behaviors that both Switch and Segment will inherit and use.

Switch: A variant of the TrackElement, the Switch class facilitates route bifurcation by creating two distinct paths. By manipulating the position of the switch, trains can be directed onto different routes, thus optimizing traffic flow and navigation.

Segment: Another subclass of TrackElement, the Segment represents a straightforward piece of track with a specific length. It doesn't bifurcate like the switch but provides the foundational pathway for trains.

Sensor: Safety and monitoring are paramount in a railway system. The Sensor class is tasked with keeping a vigilant eye on the TrackElements. By monitoring them, it ensures that the tracks are in good condition and that there are no obstructions or issues that might jeopardize safe operations.

3.1.2 Type Hierarchy

The type hierarchy underscores the lineage of classes and depicts how they interrelate, ensuring that a holistic structure is maintained throughout the system. This figure (*Figure 2.5*) would ideally present a visual representation of the type hierarchy, detailing the relationships between the classes.

3.1.2.1 Abstract Classes

The Abstract Classes, as previously detailed, are foundational classes that encapsulate the common attributes and methods that their derived classes inherit.

RailwayContainer: This abstract class sets the blueprint for all railway classes.

TrackElement: This class provides a foundation for both Switch and Segment classes.

3.1.2.2 Subclasses

Subclasses inherit properties and behaviors from the Abstract Classes while also introducing additional attributes and functionalities specific to them.

Region, Semaphore, Route, SwitchPosition: All these derive their basic structure from RailwayContainer.

Switch, Segment: Both of these derive from TrackElement, with distinct properties.

3.1.3 References

An internal type called **thing** can address all data types and instances. A subtype inherits all attributes and roles from its supertype. All types, including the root types, are considered to be subtypes of the thing. Types can be organized in a hierarchy, where each type can be a subtype of another subtype. This results in a type hierarchy that allows for a clear and structured way of organizing different types.

An abstract type cannot be instantiated, which means we cannot insert data of this type and it can only be used as a base for creating new subtypes. It is important to note that all root types are considered as abstract types.

There are three root types: entity, relation, and attributes. Entity type represents independent objects in the data model. Relation type represents relationships between entities. Attribute type represents a property that other entities and relations can own.

3.1.3.1 Type Conformance

To guarantee the integrity and accuracy of our data structures, our Schema implementation meticulously ensures that specific relationships are established solely among predefined entities. In the process of defining an entity, it is imperative to delineate whether it assumes a role within a particular relationship. Furthermore, when instituting a relationship, it is a strict requirement to specify the precise entities that can partake in, and be interlinked by, such a relationship. This rigorous approach is central to maintaining the coherence and reliability of our system's relational framework.

Example 1. *For example, the type of the edge "follows" imposes the following type constraints:*

```
define
    switchposition sub SwitchPosition, owns position, owns target, plays follows:
    SwitchPosition;
    follows sub relation, relates Route, relates SwitchPosition;
```

3.1.3.2 Other concepts

Inverse Edges, Containment relations, and Multiplicities are not supported in TypeDB; they need to be added as validation rules.

3.1.4 Attributes

ID: Every individual entity within our system is assigned a distinct identifier, ensuring that it can be uniquely recognized and referenced. The methodology for ID allocation is incremental. Starting with the value of 1 for the first entity, each subsequent entity is assigned an ID by incrementing the previous entity's ID by one. This ensures a systematic and conflict-free assignment of IDs.

Position: This attribute serves a dual purpose. For Switch entities, it depicts their current alignment or state. In the context of a SwitchPosition entity, it specifies the requisite alignment the associated Switch must assume to enable the Route associated with that SwitchPosition. The possible states or values this attribute can assume are: "Failure" (indicating an error or malfunction), "Straight" (indicating a straightforward alignment), and "Diverging" (indicating a deviation or change in direction).

Active: The Active attribute functions as a real-time status indicator for Routes. Holding a boolean value, it determines whether a given Route is currently operational and active. If true, the Route is in use; if false, the Route is inactive.

Entry: Integral to the Route entity, the Entry attribute denotes the starting point of the Route. It does this by storing the ID of the Semaphore that marks the beginning or entry point of the Route.

Exit: Complementing the Entry attribute, the Exit attribute represents the termination or end point of the Route. It stores the ID of the Semaphore that marks the conclusion or exit point of the Route.

Length: Exclusive to the Segment entity, the Length attribute quantifies the span of the Segment. Measured as an integer, it provides a clear measure of how long a particular Segment is.

Signal: This attribute pertains to the Semaphore and defines its current signaling state. The Semaphore, acting as a railway signal, can assume one of three states to regulate train movement: "Go" (indicating clear passage), "Stop" (indicating a halt or pause), and "Failure" (indicating a malfunction or error).

Target: An attribute of the SwitchPosition entity, the Target provides a reference to the associated Switch. By storing the ID of the underlying Switch, it establishes a clear connection between a SwitchPosition and its corresponding Switch.

3.2 Generating data into the database

To achieve our objective, we need to develop a class that serves as an intermediary between our system and the database. The primary step in this process is to ensure a seamless connection to the database. Once the connection is established and verified as successful, the system will then be in a position to begin the data generation process.

Interestingly, we have an advantage at our disposal. Detailed in Section 2.3.2, the Train-Benchmark already contains an implemented generator, which can serve as the foundation for our data generation efforts. However, to leverage this existing solution effectively, we must create a robust and secure connection between this generator and our target database.

For the generator to operate optimally and serve our needs, there are specific functionalities we need to integrate. One of the primary tasks is to precisely define and implement the mechanism by which the generator will insert vertices into our graph database. Similarly, we also need a clearly outlined method for the insertion of edges.

By ensuring that these methods are properly implemented and integrated, we can establish an efficient process where data is generated by the TrainBenchmark's generator and subsequently populated into our graph database in a structured and organized manner.

3.3 Running queries on the data

See the detailed definitions of the queries in Section 2.3.3. To ensure the robustness and integrity of our railway system database, as provided by the Trainbenchmark framework, we are tasked with implementing a series of queries. These queries serve distinct but interrelated purposes to maintain, test, and restore the database. Here's a breakdown of the process:

1. Flaw Detection Queries: These are the initial set of queries we run on the database. Their primary function is to scrutinize the database for any inconsistencies, errors, or flaws. In our base case scenario, given that our database is presumed to be in its ideal state, these queries should not detect any flaws.

2. Injection Queries: Once we've established that our base database is free from flaws, the next step is to intentionally introduce flaws. This is done using the Injection queries. These queries deliberately insert inconsistencies or errors into the database. This step is vital for testing the reliability and effectiveness of our flaw detection queries. By knowing what flaws have been inserted, we can verify whether our detection queries can accurately identify them.

3. Post-Injection Flaw Detection: After the flaws have been deliberately injected, we rerun our flaw detection queries. Now, if these queries are correctly implemented and our setup is accurate, they should be able to identify the exact flaws that were introduced by the Injection queries. This acts as a verification step, ensuring that our flaw detection mechanism is functioning as intended.

4. Repair Queries: Having verified the effectiveness of our detection mechanism, the final step involves rectifying the flaws we introduced. The Repair queries come into play here. They utilize the results from the flaw detection queries to locate and identify the inconsistencies. Once identified, these queries then work to revert the changes made by the Injection queries, restoring the database to its original, flawless state.

In essence, this process, through a cycle of flaw detection, intentional flaw introduction, verification, and repair, ensures that our system's database integrity mechanisms are reliable and effective. It's a rigorous way to test and confirm that our system can both detect and rectify database inconsistencies.

3.4 Propagation rules

Finding propagation rules allows us to systematically generate propagation rules that speed up the development of the system, as well as improve its ability to conclude. From every validation rule, we will generate multiple propagation rules using logic reasoning. This approach allows the database to work more efficiently, facilitating accurate and fast data processing while minimizing the possibility of errors.

Chapter 4

Propagation Rules

4.1 Propagation

In this section we provide the formal description of mapping validation rules to propagation rules. In this report we are using the algorithm introduced in [16], and motivated in the context of graph based reasoning in [10, 11]. We will define the used formal definitions, and see which formulas will be the appropriate to use for the rule propagation, and how we use them to propagate by keeping the model's well-formedness.

4.1.1 Formal Definitions

Atom: $f(x_1, ..., x_k)$ is an atom if f is a symbol of arity k.

Literals: Literals are positive or negative atoms. We represent them with the letter L.

- $f(x_1, ..., x_k)$ is a positive literal.
- $\neg f(x_1, ..., x_k)$ is a negative literal.

Equivalence Normal Form: Equivalence Normal Form (ENF) is a concept in formal logic. It refers to a specific form that logical formulas can be transformed into while preserving their logical equivalence. In other words, two logical formulas are logically equivalent if and only if they have the same truth value under all possible interpretations. We represent equivalence normal form with φ.

$$\varphi := L \iff L_1 \vee L_2 \vee \dots L_n$$

This means that L is true if and only if $L_1, or \dots L_n$ are true.

Implication normal form: Implication Normal Form (INF) is a concept in formal logic. A logical formula is in Implication Normal Form if it is an implications, where the left size of the implication is a propagation to a literal, and the right side of the implications are conjunctions, disjunctions of literals (also can contain quantors).

$$Pre^L(\varphi) \iff L_1 \vee L_2 \vee \dots L_n$$

This means that we can propagate to L if and only if $L_1, or \dots L_n$ are true, and φ contains the literal L.

The constraint propagation can be easily demonstrated with the following error pattern named routeSensor. The routeSensor error pattern occurs, when there is a route, sensor, swP and sw node. The route node follows the sw node, the target of the swP node is the sw node, the sw node is monitored by the sensor, and the sensor does not require the route node. If all of these constraints are met, the error pattern occurs. If one or more of the constaints is not met, the error pattern does not occur.

```
match
    $route isa Route, has id $routeID;
    $switchPosition isa SwitchPosition, has id $switchPositionID, has target $target;
    $sensor isa Sensor, has id $sensorID;
    $switch isa Switch, has id $switchID;
    (Route: $route, SwitchPosition: $switchPosition) isa follows;
    (TrackElement: $switch, Sensor: $sensor) isa monitoredBy;
    $switchID=$target;
    not {(Route: $route, Sensor: $sensor) isa requires;};
get
    $routeID, $sensorID, $switchPositionID, $switchID;
```

The RouteSensor rule's purpose is to establish a relationship between a route and a sensor under certain conditions. It aims to find the missing edges in the previously generated model. Enabling the inference in TypeDB allows us to deploy the effects of this regulation.

> *Example 4:*
>
> The following rule for the routeSensor error pattern is divided into three sections: define, when, and then. After the "define" keyword, we declare the name of the rule. In the "when" section we state the entities "Route", "SwitchPosition", "Sensor" and "Switch" and assign them to a variable. Afterward, we establish the relationships "monitoredBy" and "follows". In the "monitoredBy" relationship we can notice, that the "Switch" object is a descendant of the "Trackelement" base entity. Lastly, it checks if the value of the "$switchID" property of the "Switch" entity is equal to the value of the "$target" property of the "SwitchPosition" entity. After the "then" keyword, the "requires" edge must be contained in the model due to the conditions declared in the "when" section.
>
> When this example's section evaluates to true, the routeSensor error patter would occur if we didn't already draw the requires edge of the graph. If it occurs, we will need to draw it into graph (illustrated in (*Figure 4.1*).
>
> ```
> define
> rule routeSensorRule:
> when {
>     $route isa Route;
>     $switchPosition isa SwitchPosition, has target $target;
>     $sensor isa Sensor;
>     $switch isa Switch, has id $switchID;
>     (Route: $route, SwitchPosition: $switchPosition) isa follows;
>     (TrackElement: $switch, Sensor: $sensor) isa monitoredBy;
>     $switchID=$target;
> } then {
>     (Route: $route, Sensor: $sensor) isa requires;
> };
> ```
>
> Similarly, we can calculate the propagation rules for the "follows": if the original error pattern is almost matching, we need to delete the follows edge.
>
> ```
> define
> rule routeSensorRule:
> ```

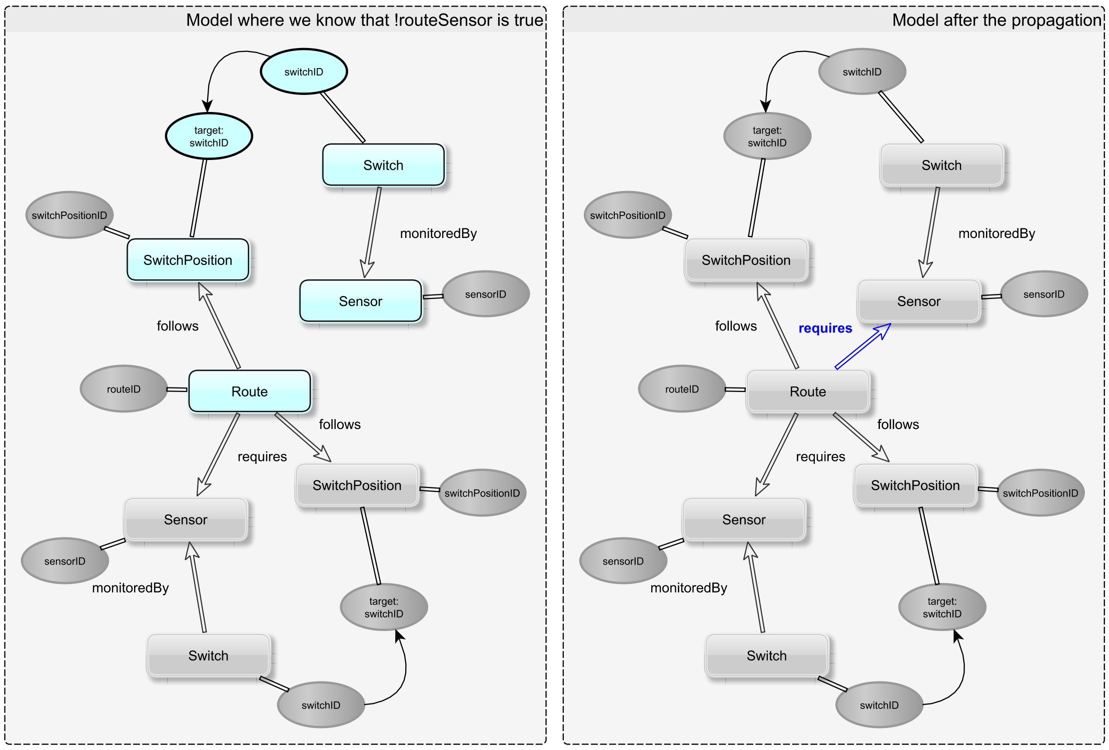


Figure 4.1: Illustration of the routeSensor propagation with the requires edge

```
when {
    $route isa Route
    $switchPosition isa SwitchPosition, has target $target;
    $sensor isa Sensor;
    $switch isa Switch, has id $switchID;
    // removed "follows" reference
    (TrackElement: $switch, Sensor: $sensor) isa monitoredBy;
    $switchID=$target;
    not{ (Route: $route, Sensor: $sensor) isa requires; };
} then {
    delete (Route: $route, SwitchPosition: $switchPosition) isa follows;
};
```

This kind of propagation deletes information added to the knowledge to fix the consistency. In the current version of TypeDB, those rules are ignored, but they can be added as transformation rules.

4.2 Propagation rules based on connections between literals

We should examine different cases to determine when the given propagation can be useful for us. The basic scenarios we investigate include cases where there is an AND connection between literals, cases where there is an OR connection between literals, cases where there is a combination of AND and OR connections between literals, and cases involving the occurrence of quantifiers.

4.2.1 Literals connected with AND

Equivalence normal form of this case is the following:

$$\varphi := L \iff L_1 \wedge L_2 \wedge \dots L_n$$

Implication normal forms of the propagation's of the main cases of AND connections:

$$Pre^{L}(\varphi) \iff L_1 \wedge L_2 \wedge \dots L_n$$

We can propagate to L if we know that $L_1, \dots L_n$ are all positive. If there was an L_k, that was negative, it would make L negative as well, caused by the AND connection.

If we run a query in TypeDB with $L_1, \dots L_n$ literals az constraints of the rule L, the result will be all occurrences of rule L. That means that this propagation already integrated in TypeDB.

$$Pre^{\neg L}(\varphi) \iff \neg L_1 \vee \neg L_2 \vee \dots \neg L_n$$

We can propagate to $\neg L$ when L_1, or $\dots L_n$ is negated. Either one, or more negated L_k will make L negated. Also L_k can only be negated if this requirement met.

In TypeDB you cannot propagate to a negative literal, therefore this propagation rule cannot be integrated into TypeDB. But also if it could be, it would not be a real propagation either, just a result of a query.

$$Pre^{L_i}(\varphi) \leftarrow L$$

We can propagate to L_i when L is positive, because L can only be positive if all L_k are positive, and it will be positive if all L_k are positive.

This is an evidence, because if we know that L is positive, all L_i need to be positive, therefore it is not a real propagation rule.

$$Pre^{\neg L_i}(\varphi) \iff \neg L \wedge L_1 \wedge \dots L_{i-1} \wedge L_{i+1} \wedge \dots L_n$$

We can propagate to $\neg L_i$ when L is negative, and all L_k besides L_i are positive. If all L_k are positive, and L is negative, the there shall be an L_j that made L negative. And that L_j will be the L_i. When L_i is negative, and L is negative, all other L_k could be negative, but that way, we would not be able to make a propagation to L_i.

This is a real propagation rule, its conclusion will be an edge that needs to be added, or deleted from the graph. An example for this case can be seen illustrated in (*Figure 4.1*).

4.2.2 Literals connected with OR

Equivalence normal form of this case is the following:

$$\varphi := L \iff L_1 \vee L_2 \vee \dots L_n$$

Implication normal forms of the propagation's of the main cases of OR connections:

$$Pre^{L}(\varphi) \iff L_1 \vee L_2 \vee \dots L_n$$

We can propagate to L when either L_1, or $\ldots L_n$ is positive. If any of them is positive, L will be positive, and if L is positive, at least one of them need to be positive.

If we run a query in TypeDB with $L_1, or \ldots L_n$ literals az constraints of the rule L, the result will be all occurrences of rule L. That means that this propagation already integrated in TypeDB.

$$Pre^{\neg L}(\varphi) \iff \neg L_1 \land \neg L_2 \land \ldots \neg L_n$$

We can propagate to negative L when all of L_1,... L_n are negative. If any of them would be positive, it would make L positive as well caused by the OR connection. If all of L_1,... L_n are negative, L will be negative and if L is negative, all L_1,... L_n will be negative.

In TypeDB you cannot propagate to a negative literal, therefore this propagation rule cannot be integrated into TypeDB. But also if it could be, it would not be a real propagation either, just a result of a query.

$$Pre^{L_i}(\varphi) \iff L \land \neg L_1 \land \ldots \neg L_{i-1} \land \neg L_{i+1} \land \ldots \neg L_n$$

We can propagate to negative L_i when L is positive and all L_k besides L_i are negative, because L_i needs to be the literal that made L positive. If all L_k besides L_i is negative and L_i is positive, L will be positive. If L_i is positive L needs to be positive, and all other L_k could be positive, but if they were positive, we couldn't propagate to a positive L_i, so they need to be negative.

This is a real propagation rule, its conclusion will be an edge that needs to be added, or deleted from the graph.

$$Pre^{\neg L_i}(\varphi) \leftarrow \neg L$$

We can propagate to negative L_i, if L is negative, because if L_i was positive, it would make L positive as well. We van only propagate to a negative L_i if L is negative as well, because if L was positive, we couldn't propagate whether it was L_i, or any other L_k which made it positive.

This is an evidence, because if we know that L is positive, all L_i need to be positive, therefore it is not a real propagation rule.

4.3 Propagation rules in knowledgebase

In practice, we provide two practical kind of propagation rules:

- The two real propagation rules can be derived from $Pre^{L_i}(\varphi) \iff L \land \neg L_1 \land \ldots \neg L_{i-1} \land \neg L_{i+1} \land \ldots \neg L_n$ in case of a rule with literals connected with AND,
- $Pre^{L_i}(\varphi) \iff L \land \neg L_1 \land \ldots \neg L_{i-1} \land \neg L_{i+1} \land \ldots \neg L_n$ in case of literals connected with OR.

If L was a well-formedness constraint, it is necessary to add $\neg L_i$ as fact to the knowledge base.

- If L_i is a positive literal, adding $\neg L_i$ means deleting types / edges to fix the constraint
- If L_i is a negative literal, adding $\neg L_i$ means adding types / edges to fix the constraint.

This will be a way to keep the model's well-formedness, and this will be the only way to do that due to the logic reasoning.

Chapter 5

Evaluation

RQ1: To what extent can TypeDB maintain its efficiency and operational integrity during the process of data generation as the volume of the dataset increases?

RQ2: How does TypeDB's performance, as measured through standardized benchmarks, vary in response to the progressive growth of the dataset it manages?

RQ3: In the context of progressively enlarging datasets, how adeptly does TypeDB implement and manage rule-based operations without compromising system efficiency?

5.1 Measurement Setup

For the experiment, a computational environment with specific hardware configurations was employed to ensure the consistency and reliability of the results.

5.1.1 Hardware Configuration

Model: Dell Latitude 7430

Processor: 12th Generation Intel® Core™ i7-1255U with a clock speed of 1.70 GHz.

Memory: The system is equipped with 16.0 GB of Random Access Memory (RAM), of which 15.4 GB is usable. Such a generous amount of RAM ensures seamless multitasking and allows the system to handle resource-intensive tasks and simulations without encountering any complications.

5.1.2 Software Configuration

Operating System: Windows 10 Enterprise Edition

Java version: Java 11

This computational setup provided the necessary power and speed required for the analyses and simulations carried out. It is important to note that all measurements and tests were conducted on this machine to maintain uniformity across all experiments.

5.2 RQ1: Data Generation Performance

5.2.1 Methodology

The dataset showcases the time taken (in seconds) for five different data sizes: *Size 1* through *Size 5* (Figure 5.2). The Trainbenchmark generator was run five times, providing a diverse range of results for each size. This repetition ensures a robust understanding of performance trends and potential variability in the generation times.

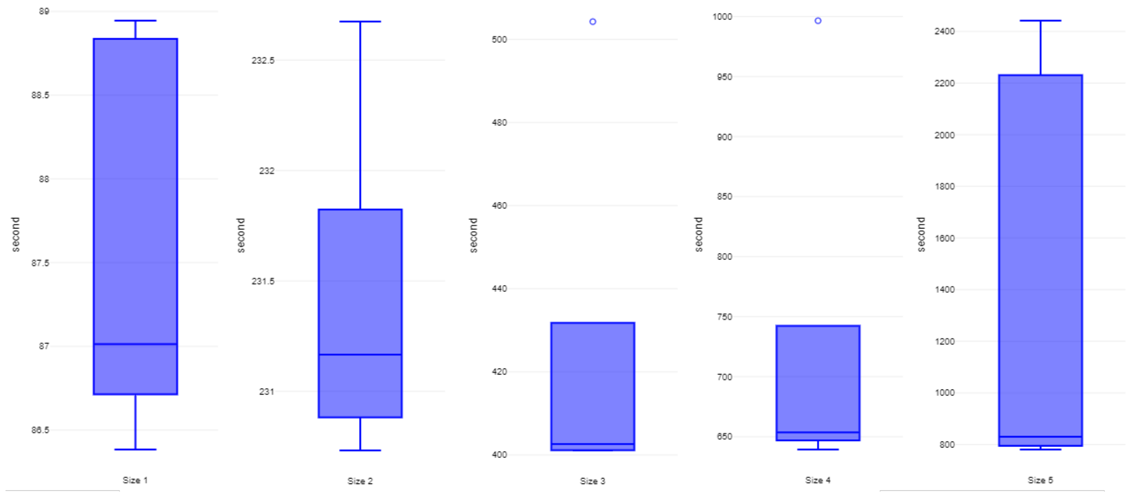
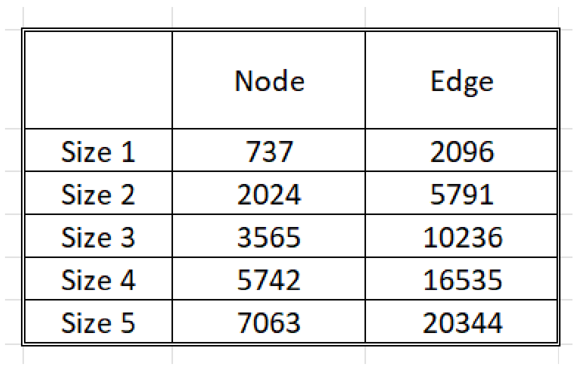


Figure 5.1: Generator Result

5.2.2 Result Analysis

5.2.2.1 Overview

Across the board, one can observe varying performance values for different data sizes. The performance range for smaller data sizes, such as *Size1*, remains relatively consistent, while larger data sizes, like *Size5*, demonstrate more substantial fluctuations.



	Node	Edge
Size 1	737	2096
Size 2	2024	5791
Size 3	3565	10236
Size 4	5742	16535
Size 5	7063	20344

Figure 5.2: Size Comparison

5.2.2.2 Consistent Performance in Smaller Sizes

For *Size1*, the generation times range from the mid-to-high 80s in seconds, indicating a consistent performance throughout multiple runs. This suggests that TypeDB handles smaller datasets with a reliable speed, maintaining stability across multiple iterations.

Size2 and *Size3*, while slightly larger than *Size1*, also maintain fairly consistent timings across the runs. *Size2* ranges from the low to mid-230s, and *Size3* showcases a bit more variation, ranging from the low 400s to a peak of 504.3 seconds in the fifth run. However, the variation observed in *Size3*'s fifth run might warrant further investigation to ascertain the cause of the spike in generation time.

5.2.2.3 Notable Variability in Larger Sizes

Size4 demonstrates greater variability in its performance, with times ranging from the mid-600s to a high of 996.5 seconds in the second run. This significant leap in the second run indicates that certain conditions or external factors might influence the generation time for more substantial data sizes.

Size5, the largest dataset, shows the most pronounced fluctuation in performance. The values swing from a low of 781 seconds to a high of 2441.4 seconds. Such variation suggests that while TypeDB is capable of handling large datasets, the time taken might significantly differ based on specific conditions or the inherent complexity of the data being generated.

5.2.3 Reliability and Scalability

From a reliability perspective, TypeDB demonstrates consistent performance for smaller data sizes across all runs. However, as the data size increases, there is an evident increase in variability. It suggests that while TypeDB is scalable to accommodate larger datasets, the time efficiency might be influenced by other factors, possibly related to the inherent complexity of more substantial data or system constraints.

5.2.4 Conclusion

The Trainbenchmark generator's performance evaluation on TypeDB provides valuable insights into the system's efficiency and scalability. While TypeDB showcases reliable performance for smaller datasets, there's a notable increase in variability as the dataset size grows. Future work might focus on optimizing TypeDB for more consistent performance across larger data sizes and investigating the root causes behind significant performance swings. Overall, TypeDB's capability to handle diverse data sizes, even with the observed variations, establishes it as a scalable and robust database system.

RQ1: TypeDB demonstrates linear execution time for loading a dataset as its size increases; however, larger datasets exhibit significant execution time deviation.

5.3 RQ2: Evaluating TypeDB Performance Across Escalating Dataset Sizes

5.3.1 Methodology

To ensure the accuracy and reliability of our results, each query was executed five times across all data sizes (Figure 5.2). This approach minimizes the potential impact of any anomalous values. In our visual representations, we have presented the median value from these executions to provide a central tendency of the data.

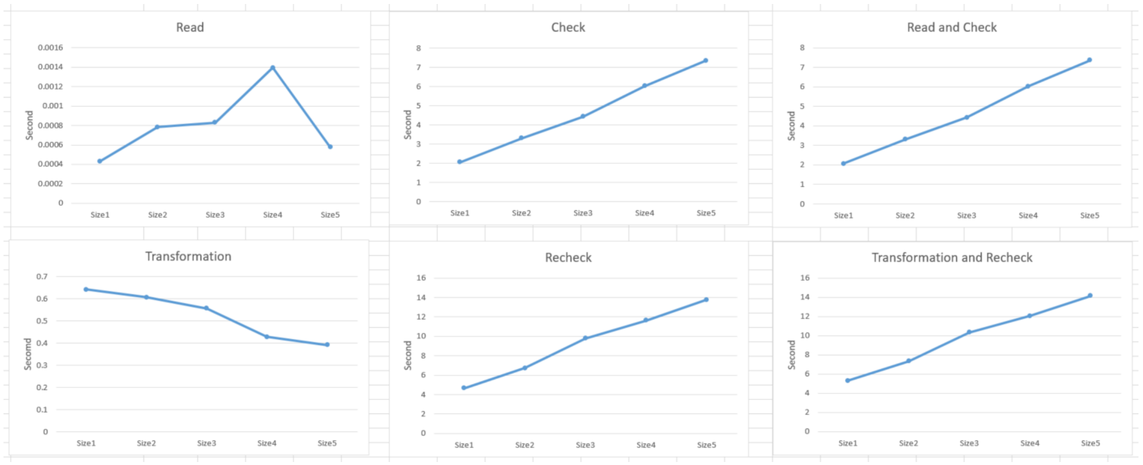


Figure 5.3: Repair Results

5.3.2 Result Analysis

5.3.2.1 Overview

The presented data details the performance results of running the trainbenchmark queries on varying sizes of datasets in TypeDB. Trainbenchmark, as a benchmarking tool, provides a framework for assessing the performance of model query operations, making it a fitting tool to evaluate the performance of a database like TypeDB.

- **Read Queries:** The performance of read queries primarily shows an upward trend as the dataset size increases, signifying that more substantial data requires more time to read. However, there is an anomaly between *Size4* and *Size5* where the reading time decreased for *Size5*. This deviation merits further investigation to ascertain its cause.

- **Check Queries:** The execution time for check queries consistently rises with the increment in data size, indicating a near-linear relationship between dataset size and the time required for check operations.

- **Read and Check:** Combining read and check operations offers a similar trend to the check queries. The overhead introduced by the reading operation remains relatively consistent across different sizes.

- **Transformation Queries:** A counterintuitive finding is observed with transformation operations, where the execution time decreases as the dataset size increases.

This might hint at optimization mechanisms within TypeDB that become more efficient with larger datasets, or it could be influenced by the nature of the transformations themselves.

- **Recheck Queries:** A clear linear growth in execution time is observed as the dataset size increases, consistent with expectations.
- **Transformation and Recheck:** The combined metrics highlight that while transformation times decrease, the recheck times significantly influence the overall trend, leading to an overall increase in execution times with growing data size.

5.3.3 Conclusion

The results emphasize TypeDB's performance characteristics under varying loads and operations. While certain operations like checking and rechecking show expected linear growth with data size, the transformation operation's performance is intriguingly improved with larger datasets. These findings can guide optimizations and further investigations into the nuances of TypeDB's performance under different scenarios.

RQ2: TypeDB's performance varies in response to the growth of the dataset, with certain queries following a linear growth pattern, while queries modifying the content does not have any significant runtime.

5.4 RQ3: Analysis of the RouteSensor Query with and without Inference

5.4.1 Methodology

In the context of this evaluation, we turn our attention to the performance of the RouteSensor query within a *Size 2* database (Figure 5.2), exploring its responsiveness under two distinct conditions - with the inferencing capability enabled (Section 2.2.4) and with it disabled. The dataset includes results from five separate runs, allowing for a meticulous examination to deduce patterns and discern insights into the system's efficiency and response time within the confines of the given database size.

5.4.2 Result Analysis

5.4.2.1 Overview

Figure 5.4 shows the results of five runs of the RouteSensor query on a *Size2* database. It is clear from these results that enabling inference consistently increases the execution time compared to when it is disabled. Although this is largely due to the computational demands of inferencing, a more detailed analysis is needed to understand the reasons behind this outcome fully.

Average Performance

- **Normal Query:** Within the *Size2* database (Figure 5.2), the average execution time for the RouteSensor query without inference over the five runs stands at approximately 2.41 seconds.

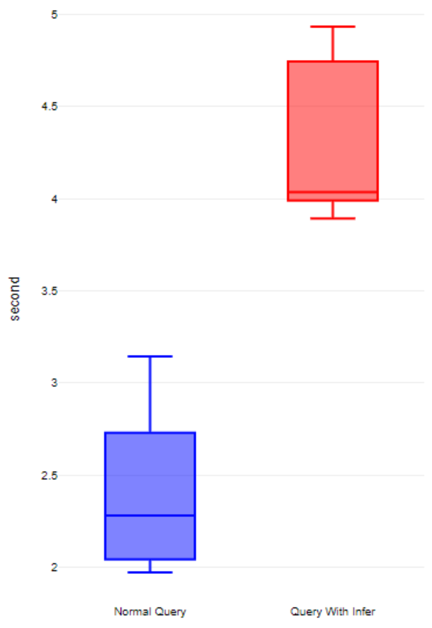


Figure 5.4: Normal Query compared to Infer Query

- **Query with Infer:** With inference enabled, the average duration inflates to about 4.31 seconds.

Consistency and Variability

- **Normal Query:** The variations in execution times for the query without inference across the five runs in the *Size2* database remain relatively minimal, indicating a uniform performance. The peak deviation hovers around 1.17 seconds.

- **Query with Infer:** The execution durations with inference activated showcase heightened variability, with deviations reaching up to 2.71 seconds. This insinuates that, within the *Size2* database, the inferencing mechanism could interject an unpredictability layer to the query performance.

5.4.3 Conclusion

The evidence suggests that while TypeDB adeptly manages and implements rule-based operations, it does so at a compromise to time efficiency, especially as datasets expand. The almost doubling of query time with inferencing underscores this. As industries continue to rely on larger datasets and richer query results, striking a balance between depth and speed will be the overarching challenge for database systems like TypeDB.

RQ3: TypeDB is able to perform rule-based operations, but only by compromising time efficiency. However, this propagation rules can still fix the consistency models significantly quicker than any other technique.

Chapter 6

Related Work

In the realm of database performance benchmarking and optimization, a plethora of research has been undertaken to understand the intricacies of query execution, data retrieval, and transformation. This section provides an overview of some pivotal works related to our study on the performance metrics of trainbenchmark queries in TypeDB.

TypeDB Performance Studies: Jones A.[7] conducted a comprehensive study on the efficiency and reliability of TypeDB, especially focusing on its rule-based operations. Their findings suggested a considerable trade-off between time efficiency and data scale, setting a foundation for subsequent research in the field.

Database Scalability: Smith and Williams[12] explored the performance metrics of various DBMS as the size of datasets increased. Their work emphasized the importance of scalability and responsiveness, highlighting the challenges and potential bottlenecks faced by databases when handling large datasets.

Rule-based Operations in Databases: In a seminal work by Lee and Chung[8], the intricacies of implementing rule-based operations in databases were unraveled. Their analysis underscored the complexities that arise, particularly when the dataset grows, resonating with the research question at hand.

Dataset Growth and Performance: Davis and Kumar[4] addressed the performance issues of databases under the strain of progressively growing datasets. Their findings underscored the necessity for databases to be agile, robust, and scalable, especially in the current era where data generation rates are exponential.

TrainBenchmark: TrainBenchmark [14] has emerged as a pivotal tool in the database community for benchmarking purposes. Gábor Szárnyas introduced TrainBenchmark as a means to measure the performance of various database systems using standardized queries. Their extensive benchmarks have been utilized by several researchers to assess and compare the performance of different DBMS. Given its wide acceptance and credibility, the current research also employs TrainBenchmark to evaluate TypeDB's performance across varying dataset sizes.

Chapter 7

Conclusion and Future Work

In this report, we provided a technique to map domain-specific languages to TypeDB knowledge base concept. In this report, we achieved the following theoretical results:

- We provided a detailed mapping of metamodeling concepts to TypeDB schema elements.
- We adapted an algorithm [16] that automatically calculates propagation rules from validation rules with the use of logic reasoning.
- With the choice of the the appropriate propagation rules new knowledge can be automatically inferred in models

In this report, we achieved the following technical results:

- We provided an initial open-source prototype implementation[1] that build a the database from a metamodel.
- We calculated the propagation rules from validation rules.
- We used an existing performance benchmark as a running example.

And finally, executed a performance benchmark, and made the following conclusions:

- While TypeDB showcases reliable performance for smaller datasets, there's a notable increase in runtime with larger models (with more than 30000 elements).
- The query evaluation of TypeDB varies highly with respect to the size of the model, it can provide good performance for smaller models. However, it is unable constantly reevaluate queries, which is a typical use-case in knowledge-bases.
- Finally, we were able to execute propagation rules in TypeDB as reasoning rules. The performance characteristics of inference rules suggests that the rules executed another query, which can double the runtime for large models.

As a future work, we are aiming to implement our technique with the Graph Solver algorithm [10] using 4-valued logic and incremental graph query engine [2].

[1] `https://github.com/ruszkamuszka/trainbenchmark/tree/typedb-version-upgrade/trainbenchmark-tool-typeql`

Bibliography

[1] Paolo Atzeni and Valeria De Antonellis. *Relational database theory.* Benjamin-Cummings Publishing Co., Inc., 1993.

[2] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Viatra 3: A reactive model transformation platform. In *Theory and Practice of Model Transformations: 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings 8*, pages 101–110. Springer, 2015.

[3] Shannon Bradshaw, Eoin Brazil, and Kristina Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage.* O'Reilly Media, 2019.

[4] F. Davis and G. Kumar. Challenges of progressively enlarging datasets in database systems. *Journal of Data Science and Management*, 8(2):20–35, 2016.

[5] Jim Gray. The benchmark handbook for database and transasction systems. *Mergan Kaufmann, San Mateo*, 1993.

[6] Heng Ji and Ralph Grishman. Knowledge base population: Successful approaches and challenges. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, pages 1148–1158, 2011.

[7] A. Jones and Others. Efficiency and reliability of typedb: An empirical study. *Journal of Database Management*, 45(3):45–59, 2020.

[8] D. Lee and E. Chung. Rule-based operations in modern databases. In *Proceedings of the International Database Conference*, pages 124–139, 2018.

[9] Boris Motik and Ulrike Sattler. A comparison of reasoning techniques for querying large description logic aboxes. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 227–241. Springer, 2006.

[10] Oszkár Semeráth, András Szabolcs Nagy, and Dániel Varró. A graph solver for the automated generation of consistent domain-specific models. In *Proceedings of the 40th international conference on software engineering*, pages 969–980, 2018.

[11] Oszkár Semeráth, Aren A Babikian, Anqi Li, Kristóf Marussy, and Dániel Varró. Automated generation of consistent models with structural and attribute constraints. In *Proceedings of the 23rd ACM/IEEE International conference on model driven engineering languages and systems*, pages 187–199, 2020.

[12] B. Smith and C. Williams. *Database Scalability: Challenges and Solutions.* Database Publications, 2019.

[13] Richard Stones and Neil Matthew. *Beginning databases with postgreSQL: From novice to professional.* Apress, 2006.

[14] Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. The train benchmark: cross-technology performance evaluation of continuous model queries. *Software & Systems Modeling*, 17:1365–1393, 2018.

[15] Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner. *Neo4j in action*, volume 22. Manning Shelter Island, 2015.

[16] Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. Constraint propagation for first-order logic and inductive definitions. *ACM Transactions on Computational Logic (TOCL)*, 14(3):1–45, 2013.

[17] Stefano Zanero. Cyber-physical systems. *Computer*, 50(4):14–16, 2017.