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

Incremental Graph Queries in the Cloud

Scientific Students' Associations Report

Author
Gábor Szárnyas

Supervisors
Dr. István Ráth
Benedek Izsó
Dr. Dániel Varró

October 19, 2013

Kivonat

Az adatintenzív alkalmazások nagy kihívása a lekérdezések hatékony kiértékelése. A modellvezérelt szoftvertervezés (MDE) során az eszközök és a transzformációk különböző bonyolultságú lekérdezéssekkel dolgoznak. Míg a szoftvermodellek mérete és komplexitása folyamatosan nő, a hagyományos MDE eszközök gyakran nem skálázódnak megfelelően, így csökkentve a fejlesztés produktivitását és növelve a költségeket.

Ugyan az újgenerációs, ún. NoSQL adatbázis-kezelő rendszerek többsége képes horizontális skálázhatóságra, az ad-hoc lekérdezéseket nem támogatja olyan hatékonyan, mint a relációs adatbázisok. Mivel a modellvezérelt alkalmazások tipikusan komplex lekérdezéseket futtatnak, a NoSQL adatbázis-kezelők közvetlenül nem használhatók ilyen célra.

Diplomatervem célja, hogy az EMF-IncQuery-ben alkalmazott inkrementális gráfmintaillesztő algoritmust elosztott, felhőalapú infrastruktúrára implementáljam. Az IncQuery-D prototípus skálázható, így képes több számítógépből álló fürtön nagy modelleket kezelni és komplex lekérdezések hatékonyan kiértékelni. Az elképzelés életképességét előzetes mérési eredményeink igazolják.

Abstract

Queries are the foundations of data intensive applications. In model-driven software engineering (MDE), model queries are core technologies of tools and transformations. As software models are rapidly increasing in size and complexity, traditional MDE tools frequently exhibit scalability issues that decrease productivity and increase costs.

While such scalability challenges are a constantly hot topic in the database community and recent efforts of the NoSQL movement have partially addressed many shortcomings, this happened at the cost of sacrificing the powerful ad-hoc query capabilities of SQL. Unfortunately, this is a critical problem for MDE applications, as their queries can be significantly more complex than in general database applications.

In my thesis work, I aim to address this challenge by adapting incremental graph search techniques – known from the EMF-INCQUERY framework – to the distributed cloud infrastructure. INCQUERY-D, my prototype system can scale up from a single-node tool to a cluster of nodes that can handle very large models and complex queries efficiently. The feasibility of my approach is supported by early experimental results.

Contents

Abstract in Hungarian **3**

Abstract in English **4**

1 Introduction **8**
1.1 Context 8
1.2 Problem statement, requirements 9
1.3 Objectives 9
1.4 Contribution, added value 9
1.5 Structure of the report 9

2 Background technologies **10**
2.1 Big Data and the NoSQL movement 10
2.2 Graph models 11
2.2.1 Graph models 11
2.2.2 Triplestores 13
2.3 NoSQL technologies 13
2.3.1 Cassandra 13
2.3.2 Hadoop 13
2.3.3 Neo4j 15
2.4 Graph technologies 15
2.4.1 Titan 15
2.4.2 4store 18
2.5 Asynchronous messaging 18
2.5.1 Akka 19
2.6 Google Guava libraries 19
2.7 Eclipse-based technologies 20
2.7.1 EMF 20
2.7.2 EMF-IncQuery 20

3 Overview **22**

3.1 Case study: TrainBenchmark 22
3.1.1 Domain 23
3.1.2 Queries 23
3.2 Architecture overview 25
3.2.1 Rete in general 25
3.2.2 IncQuery-D architecture 25
3.3 Initialization and indexing 26
3.3.1 Indexing 26
3.3.2 Graph-like data manipulation 26
3.3.3 Notification mechanisms 27
3.4 Incremental queries, change propagation 27
3.4.1 Rete 27
3.4.2 Distributed operation 28
3.4.3 Scalability considerations 28
3.5 Deployment, configuration 30
3.5.1 Tooling 30
3.5.2 Degrees of freedom 30
3.5.3 Workflow 30
3.6 Elaboration of the example 32
3.6.1 Workflow 32

4 Evaluation **36**
4.1 Goals 36
4.1.1 Phases 36
4.1.2 Measure the response time and the scalability 36
4.1.3 Workload profile's difference from standard benchmarks 36
4.2 Benchmark scenario 36
4.3 Generation of models 37
4.4 Benchmark environment 38
4.4.1 Benchmark setup 38
4.4.2 Hardware, software ecosystem 39
4.4.3 Benchmark methodology 39
4.4.4 Data collection 39
4.4.5 Data processing tools 39
4.5 Results 40
4.5.1 Results using the Neo4j graph database 40

5 Related and future work **44**
5.1 Related work 44

5.2 Future work 45

6 Conclusions **46**

6.1 Summary of contributions 46

6.1.1 Own work 46

6.2 Limitations 47

6.3 Future directions 47

Bibliography **48**

A Graph formats **52**

A.1 Property graph formats 52

A.1.1 GraphML 52

A.1.2 Blueprints GraphSON 53

A.1.3 Faunus GraphSON 54

A.2 Semantic graph formats 54

A.2.1 RDF 54

Chapter 1

Introduction

Nowadays, model-driven software engineering (MDE) plays an important role in the development processes of critical embedded systems. Advanced modeling tools provide support for a wide range of development tasks such as requirements and traceability management, system modeling, early design validation, automated code generation, model-based testing and other validation and verification tasks. With the dramatic increase in complexity that is also affecting critical embedded systems in recent years, modeling toolchains are facing scalability challenges as the size of design models constantly increases, and automated tool features become more sophisticated.

1.1 Context

Many scalability issues can be addressed by improving query performance. *Incremental evaluation* of model queries aims to reduce query response time by limiting the impact of model modifications to query result calculation. Such algorithms work by either (i) building a cache of interim query results and keeping it up-to-date as models change (e.g. EMF-INCQUERY [23]) or (ii) applying impact analysis techniques and re-evaluating queries only in contexts that are affected by a change (e.g. the Eclipse OCL Impact Analyzer [30]). This technique has been proven to improve performance dramatically in several scenarios (e.g. on-the-fly well-formedness validation or model synchronization), at the cost of increasing memory consumption. Unfortunately, this overhead is combined with the increase in model sizes due to in-memory representation (found in state-of-the-art frameworks such as EMF [45]). Since single-computer heaps cannot grow arbitrarily (as response times degrade drastically due to garbage collection problems), memory consumption is the most significant scalability limitation.

An alternative approach to tackling MDE scalability issues is to make use of advances in persistence technology. As the majority of model-based tools uses a graph-oriented data model, recent results of the NoSQL and Linked Data movement [39, 1, 2] are straightforward candidates for adaptation to MDE purposes. Unfortunately, this idea poses difficult conceptual and technological challenges: (i) property graph databases lack strong metamodeling support and their query features are simplistic compared to MDE needs, and (ii) the underlying data representation format of semantic databases (RDF [31]) has crucial conceptual and technological differences to traditional metamodeling languages such as Ecore [45]. Additionally, while there are initial efforts to overcome the mapping issues between the MDE and Linked Data worlds [34], even the most sophisticated NoSQL storage technologies lack efficient and mature support for executing expressive queries incrementally.

1.2 Problem statement, requirements

[42]

1.3 Objectives

We aim to address these challenges by adapting incremental graph search techniques from EMF-INCQUERY to the cloud infrastructure. We introduce INCQUERY-D, a prototype system based on a distributed Rete network [28] that can scale up from a single-workstation tool to a cluster to handle very large models and complex queries efficiently (**??**). We carry out an initial performance evaluation in the context of on-the-fly well-formedness validation of software design models (**??**), discuss related work in section 5.1 and conclude the thesis in **??**.

1.4 Contribution, added value

1.5 Structure of the report

Chapter 2

Background technologies

Implementing a scalable graph pattern matcher requires a wide range of technologies. Careful selection of the technologies is critical to the project's success. For INCQUERY-D, we were looking for technologies that were designed with scalability in mind and have been deployed in large-scale distributed systems successfully. To avoid licensing issues and costs, our search criteria included that the technologies must be free and open-source solutions.

2.1 Big Data and the NoSQL movement

Since the 1980s, database management systems based on the relational data model [25] dominated the database market. Relational databases have a number of important advantages: precise mathematical background, understandibility, mature tooling and so on. However, due to the strongly connected nature of their data model, relational databases often have scalability issues [43].

In the last decade, organizations often struggled to store and process the huge amounts of data they produced. This problem introduces a diverse palette of scientific and engineering challenges, often called *Big Data* challenges.

Big Data challenges spawned dozens of new database management systems. Typically, these systems broke with the strictness of the relational data model and utilized simpler, more scalable data models. These systems dropped support for the SQL query language used in relational databases and hence were called *NoSQL databases*[1] [14]. During the development of INCQUERY-D's propotype, we experimented with numerous NoSQL databases.

[1] The community now mostly translates NoSQL as ”not only SQL”.

2.2 Graph models

The graph is a well-known mathematical concept widely used in computer science. For our work, it's important to distinguish between different graph data models.

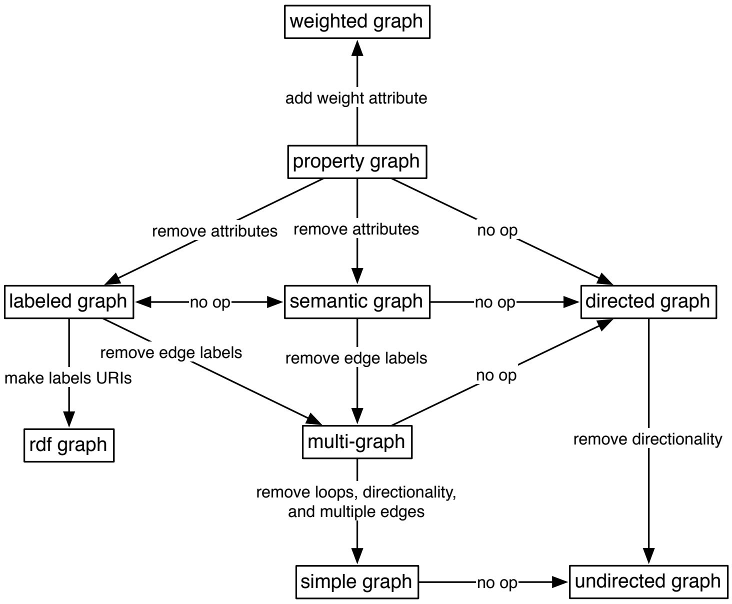

Figure 2.1: *Different graph data models [41]*

2.2.1 Graph models

The most basic graph model is the *simple graph*, formally defined as $G = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. Simple graphs are sometimes referred as textbook-style graphs because they are an integral part of academic literature. Simple graphs are useful for modeling homogeneous systems and have plenty of algorithms for processing.

Simple graphs can be extended in several different ways (Figure 2.1). To describe the connections in more detail, we may add directionality to edges (*directed graph*), allow loops and multiple edges (*multi-graph*). To allow different connections, we may label the edges (*labeled graph* or *semantic graph*). *RDF graph*s use URIs instead of labels, otherwise they have similar expressive power as labeled graphs. *Property graph*s add even more possibilites by introducing properties. Each graph element, both vertices and edges can be described with a collection of properties. The properties are key–value pairs, e.g. `type = 'Person'`, `name = 'John'`, `age = 34`. Property graphs are powerful enough to describe Java objects or EMF instance models (subsection 2.7.1).

TinkerPop

The *TinkerPop* framework is an open-source software stack for graph storage and processing. TinkerPop includes Blueprints, a property graph model interface. Blueprints fulfills the same role for graph databases as JDBC does for relational databases. Most NoSQL graph databases implement the property graph interface provided by Blueprints, including Neo4j (subsection 2.3.3), Titan (subsection 2.4.1), DEX [17], InfiniteGraph [15] and OrientDB [16].

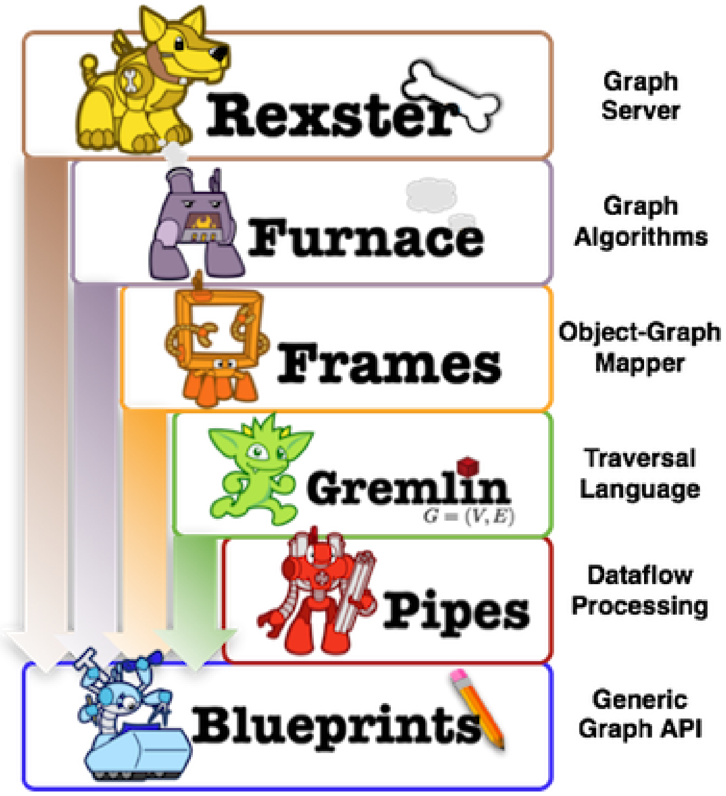

Figure 2.2: *The TinkerPop software stack*

TinkerPop also introduces a graph query language, *Gremlin*. Gremlin is a domain-specific language based on Groovy, a Java-like dynamic language which runs on the Java Virtual Machine. Unlike most query languages, Gremlin is an imperative language with a strong focus on graph traversals. Gremlin is based on *Pipes*, TinkerPop's dataflow processing framework. Besides the traversing, Gremlin capable of analyzing and manipulating the graph as well.

TinkerPop also provides a graph server (*Rexster*), a set of graph algorithms tailored for property graphs (*Furnace*) and an object-graph mapper (*Frames*). The TinkerPop software stack is shown on Figure 2.2.

2.2.2 Triplestores

Triplestores are tailored to store and process triples efficiently. A triple is a data entity composed of a subject, a predicate and an object, e.g. "John instanceof Person", "John is 34". Triplestores are mostly used in semantic technology projects. Also, some triplestores are capable of *reasoning*, i.e. inferring logical consequences from a set of facts or axioms.

Triplestores use the RDF (Resource Description Framework) data model. Although the RDF data model has less expressive power than the property graph, by introducing additional resources for each property, a property graph can be easily mapped to a RDF. Triplestores are usually queried via the RDF format's query language, SPARQL.

2.3 NoSQL technologies

In the following section we summarize the core technologies used in INCQUERY-D's prototype implementation. We briefly introduce the goals of each technology, with particular emphasis on the scalability aspects.

2.3.1 Cassandra

Cassandra is one of the most widely used NoSQL databases [6]. Originally developed by Facebook [36], Cassandra is now an open-source Apache project.

Cassandra's data model is called *column family*. A column family is similar to a table of a relational database: it consists of rows and columns. However, unlike in a relational database's table, the rows do not have to have the same fixed set of columns. Instead, each row can have a different set of columns. This makes the data structure more dynamic and avoids ~~the plethora of~~ problems associated with NULL values.

To distribute the data across the cluster, Cassandra uses a partitioner mechanism. The simplest partitioners distribute the rows evenly based on their hash values. Currently, Cassandra provides *RandomPartitioner* (MD5 hash) and *Murmur3Partitioner* (Murmur3 hash).

Cassandra has sophisticated fault-tolerance mechanism. It allows the application to balance between availability and consistency by allowing it to tune the consistency constraints. Cassandra is written in Java.

2.3.2 Hadoop

Hadoop is an open-source, distributed data processing framework inspired by Google's publications about MapReduce [26] and the Google File System [29]. Originally developed at Yahoo!, Hadoop is now an Apache project [7]. Like Google's systems, Hadoop is designed to run on commodity hardware, i.e. server clusters built from commercial off-the-shelf products. Hadoop provides a distributed file system (HDFS) and a column family database (HBase). All software in the Hadoop framework is written in Java.

The MapReduce paradigm defines a parallel, asynchronous way of processing the data. As the name implies, MapReduce consists of two phases: the *map* function processes each item of a list. The resulted list is then aggregated by the *reduce* function.

A typical small Hadoop cluster consists of a single master node which is responsible for the coordination of the cluster and worker nodes which deal with the data processing. The MapReduce job is coordinated by the master's *job tracker* and processed by the slave nodes' *task tracker* modules (Figure 2.3).

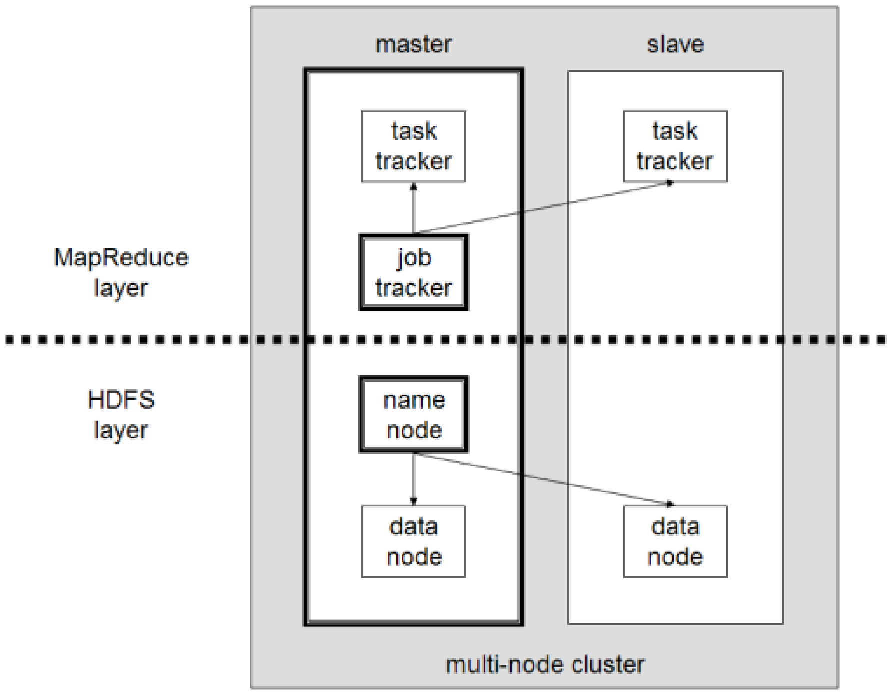

Figure 2.3: *Hadoop's architecture*

HDFS

The Hadoop Distributed File System (HDFS) is an open-source, distributed file system, inspired by the Google File System and written specifically for Hadoop [7]. Unlike other distributed file systems (e.g. Lustre [12]), which require expensive hardware components, HDFS was designed to run on commodity hardware.

HDFS tightly integrates with Hadoop's architecture (Figure 2.4). The *NameNode* is responsible for storing the metadata of the files and the location of the replicas. The data is stored by the *DataNode*s.

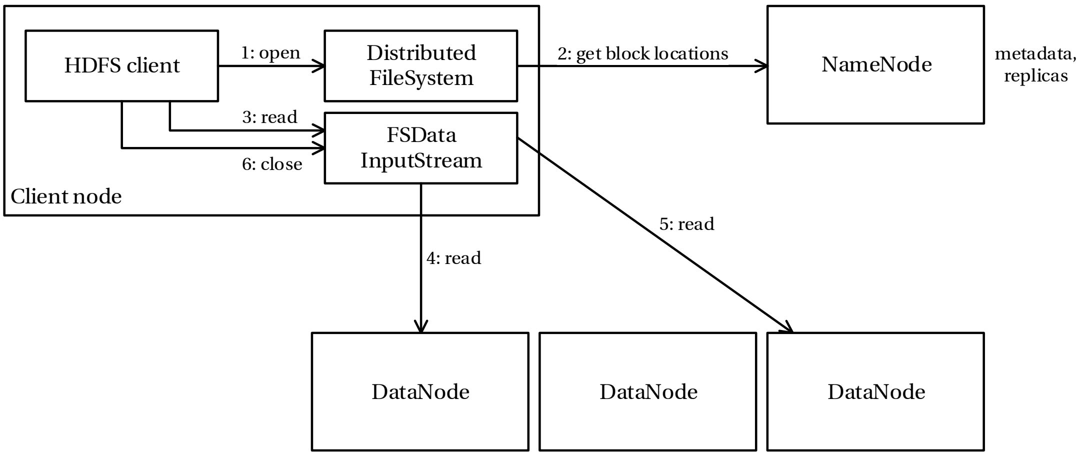

Figure 2.4: *HDFS' architecture*

HBase

HBase is an open-source, distributed column family database. It is developed as part of the Hadoop project and runs on top of HDFS. The tables in an HBase database can serve as the input and the output for MapReduce jobs run in Hadoop.

2.3.3 Neo4j

Neo4j, developed by Neo Technology, is the most popular NoSQL graph database. Neo4j implements TinkerPop's Blueprints property graph data model along with Gremlin. It also provides Cypher, a declarative query language for graph pattern matching.

Neo4j is one of the most mature NoSQL databases. It's well documented and provides ample tooling. However, it's scalability features are limited: instead of sharding, it only supports replication of data to create a highly available cluster. Of course, the scalability limitations are a hot topic in Neo4j's development. Neo4j's developers make serious efforts to improve Neo4j's scalability in an ongoing project called Rassilon [3].

Neo4j is capable of loading graphs from GraphML [18] and Blueprints GraphSON [10] formats (see section A.1 for examples). Neo4j graphs can be visualized in Neoclipse, an Eclipse RCP application [13]. A part of a TrainBenchmark (section 3.1) instance model is shown on Figure 2.5.

2.4 Graph technologies

2.4.1 Titan

Titan is an open-source, distributed, scalable graph database from Aurelius, the creators of the TinkerPop framework. Unlike Neo4j, Titan is not a standalone database. In-

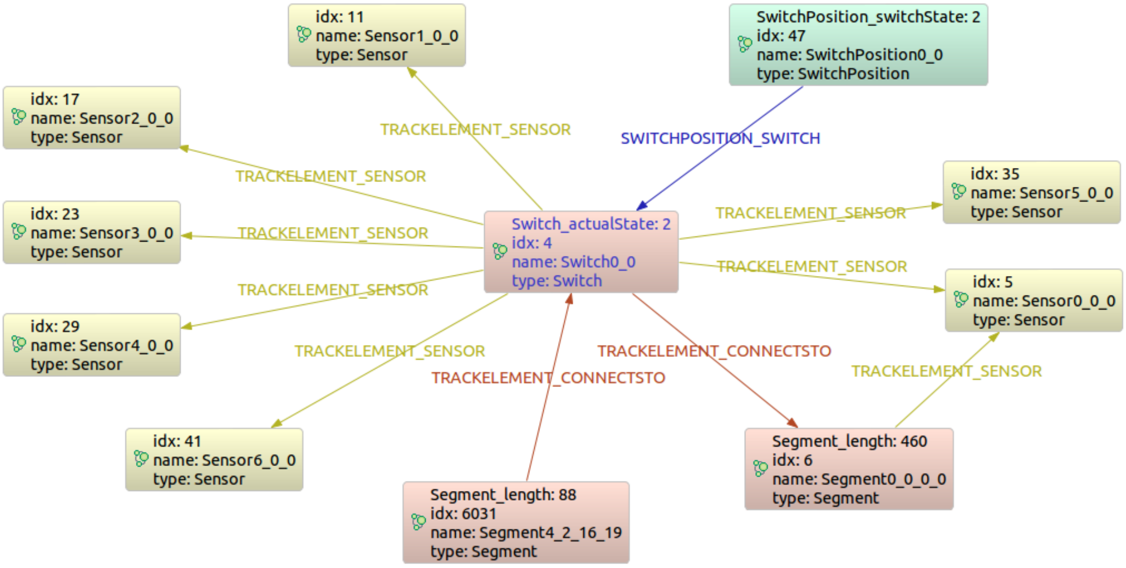

Figure 2.5: *TrainBenchmark subgraph visualized in Neoclipse*

stead, it builds on top of existing NoSQL database technologies and leverages Hadoop's MapReduce capabilities. Titan supports various storage backends, including Cassandra and HBase.

Mapping and sharding

To store the graph, Titan maps each vertex to a row of a column family (Figure 2.6). The row stores the identifer and the properties of the vertex, along both the incoming and outgoing edges' identifiers, labels and properties.

Titan uses the storage backend's partitioner (e.g. Cassandra's RandomPartitioner) to shard the data. A more sophisticated partitioning system that will allow for partitioning based on the graph's static and dynamic properties (its domain and connectivity, respectively) is under implementation, but not yet available.

Deployment

Titan can be deployed in different ways according to the needs of the application. For INCQUERY-D's prototype, we used Titan in *remote server mode* (Figure 2.7). In this setup, Titan runs in the same Java Virtual Machine as the application and communicates with the Cassandra cluster on a low-level protocol (e.g. Thrift).

Faunus

Although Titan was designed with scalability in mind, it's query engine does not work in a parallel fashion. Also, it's unable to cope with queries resulting in millions of graph

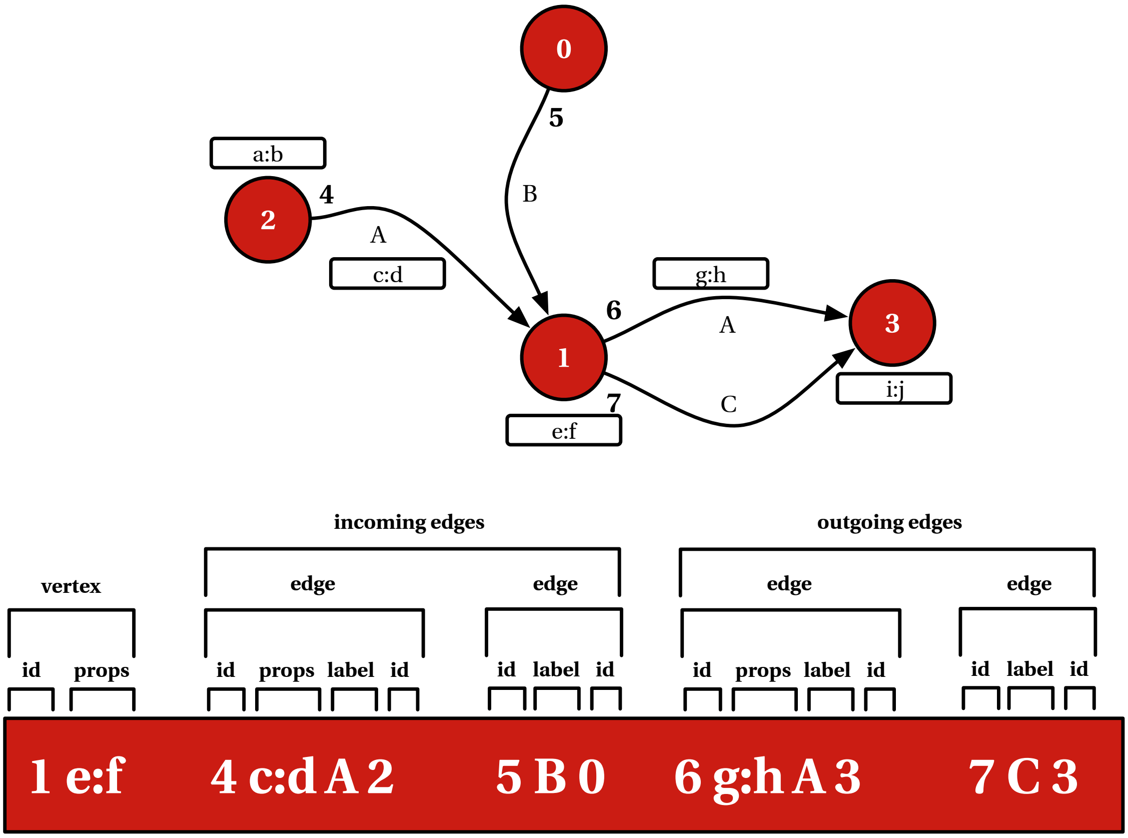

Figure 2.6: *Titan graph vertex stored in Cassandra as a row*

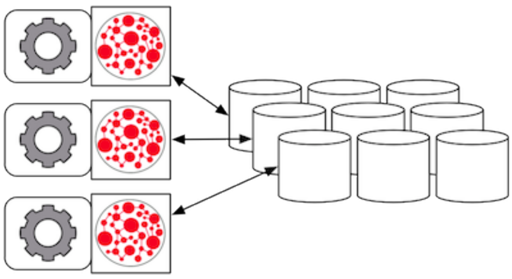

Figure 2.7: *Using Titan with Cassandra in remote server mode*

elements. To address this shortcoming, Aurelius developed a Hadoop-based graph analytics engine, Faunus.

Faunus has its own format called Faunus GraphSON. The Faunus GraphSON format is vertex-centric: each row represents a vertex of the graph. This way, Hadoop is able to efficiently split the input file and parallelize the load process. See subsection A.1.3 for an example.

It's important to note that Faunus always traverses the whole graph and does not use its indices. This makes it slow for retrieving nodes or edges by type (see our typical workload at XXX).

2.4.2 4store

4store is an open-source, distributed triplestore created by Garlik [4]. Unlike the other tools discussed earlier, 4store is written in C. While 4store is primarily applied for semantic web projects, it's maturity and scalability ~~made it an appropriate storage back-end for INCQUERY-D's prototype.~~

Similar to Titan's partitioning, 4store's sharding mechanism (called *segmenting*) distributes the RDF resources evenly across the cluster. However, unlike Titan, 4store's data model is an RDF graph. Hence, 4store's input format is RDF/OWL.

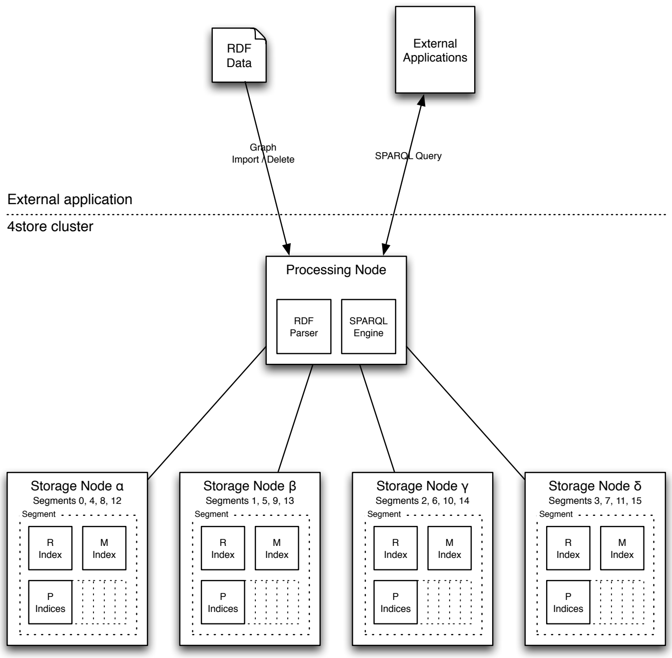

Figure 2.8: *4store's distributed architecture [32]*

Table 2.1: *Overview of database technologies*

Technology	Data model	Sharding	Distributed operation	DML facility	Identifier generation
Neo4j	Property graph	Manual	Manual	Cypher	Manual
4store	RDF	Automatic	Manual	SPARQL	Manual
Titan	Property graph	Automatic	Automatic	Gremlin	Automatic

2.5 Asynchronous messaging

Most distributed, concurrent systems use a messaging framework or message queue service. Because of the nature of the Rete algorithm (3.2.1), INCQUERY-D requires a distributed, asynchronous messaging framework.

2.5.1 Akka

Akka is an open-source, fault-tolerant, distributed, asynchronous messaging framework developed by Typesafe. Akka is implemented in Scala, a functional and object-oriented programming language which runs on the Java Virtual Machine. Akka provides language bindings for both Java and Scala.

Akka provides strong support for remoting. Actors have both a logical and a physical path (Figure 2.9). This way, they can be transparently moved between nodes of the cluster.

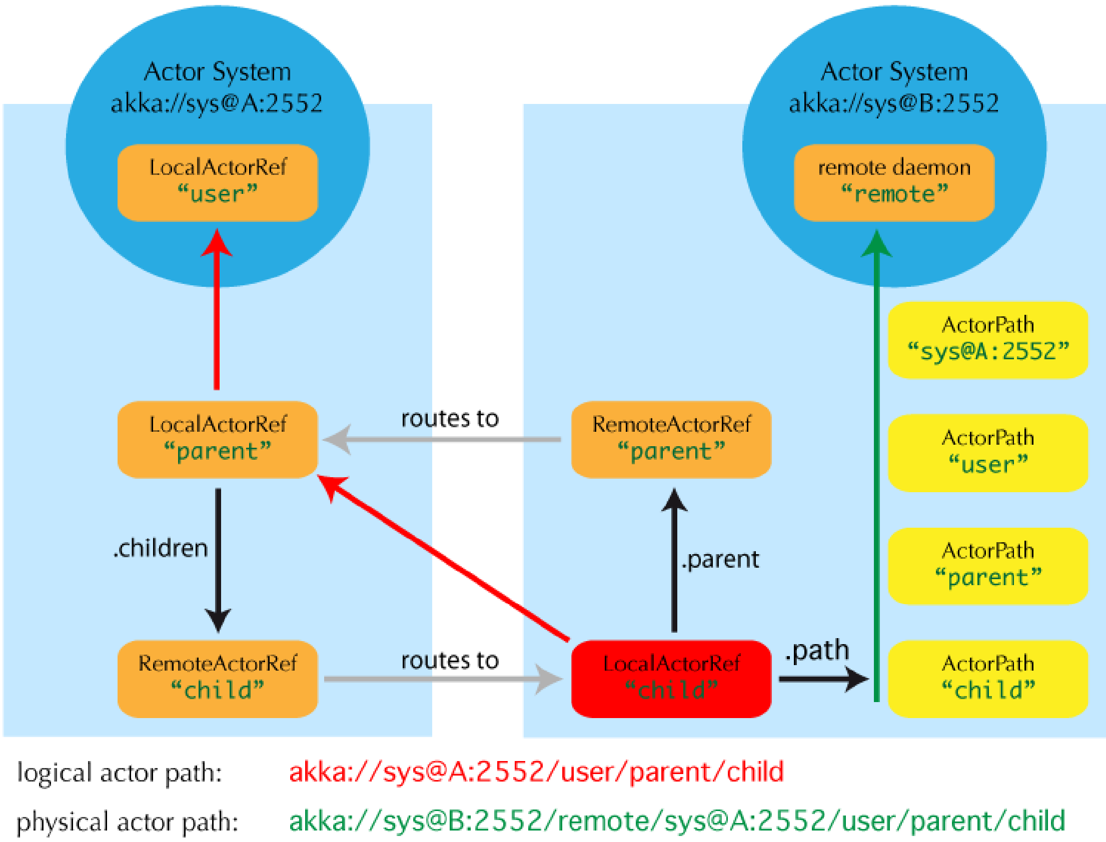

Figure 2.9: *Deploying a remote actor in Akka*

2.6 Google Guava libraries

INCQUERY-D relies heavily on Google Guava library's Collections framework [11]. The Guava Collection is an extension to Java's Collection framework. We used both immutable collections and new type of collections (e.g. the `Multimap` interface and its implementations).

2.7 Eclipse-based technologies

Eclipse is a free, open-source software development environment and a platform for plug-in development. Members of the Eclipse Foundation include industry giants like IBM, Intel, Google and SAP.

INCQUERY-D's single workstation predecessor, EMF-INCQUERY is built around Eclipse-based technologies. To reap the benefits of a mutual code base, we designed INCQUERY-D to use as much of EMF-INCQUERY's components as possible. In the following section, we introduce the Eclipse-based technologies most important for our work.

2.7.1 EMF

Eclipse comes with its own modeling technologies called EMF (Eclipse Modeling Framework). EMF provides a metamodel (Ecore) for designing applications and a code generation facility to produce the Java classes for the model.

2.7.2 EMF-INCQUERY

EMF-INCQUERY is developed by the Fault Tolerant Systems Research Group (FTSRG) in the Budapest University of Technology and Economics. EMF-INCQUERY is an open-source Eclipse project which provides incremental query evaluation on EMF models. The queries (graph patterns) are defined in INCQUERY Pattern Language (IQPL), a domain-specific language implemented in Xtext and Xtend. EMF-INCQUERY's architecture is shown on Figure 2.10.

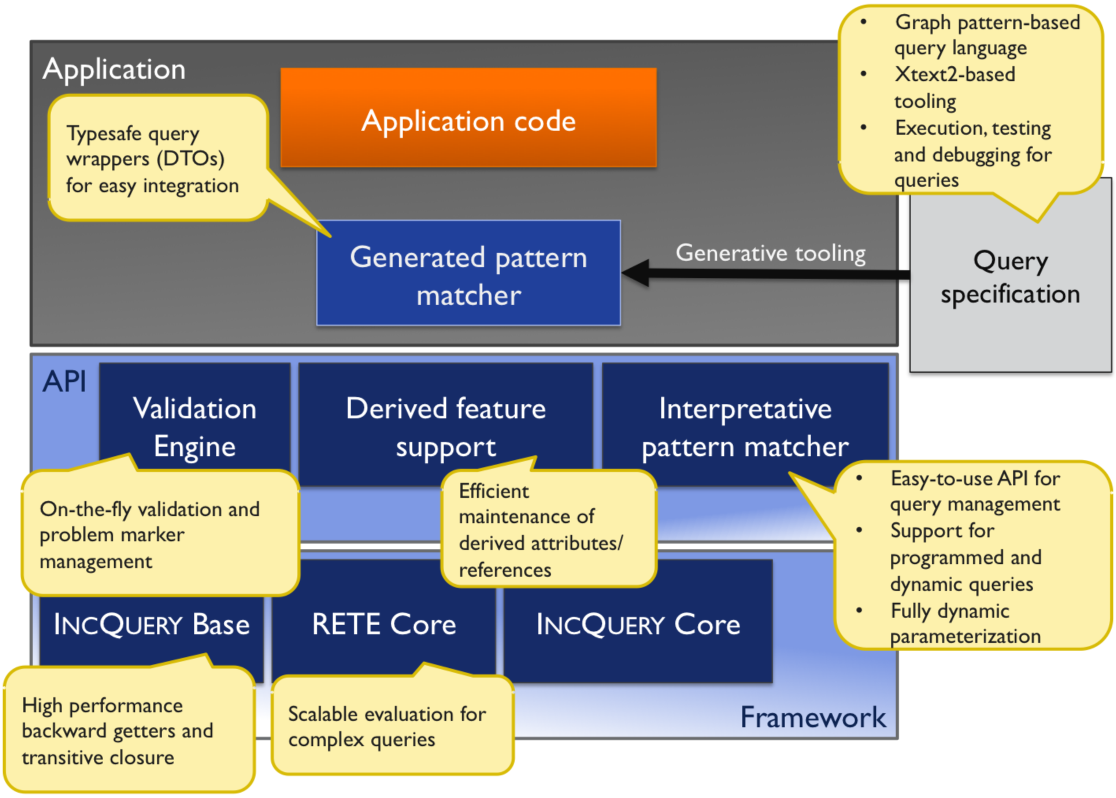

Figure 2.10: EMF-INCQUERY*'s architecture*

Chapter 3

Overview

The primary goal of INCQUERY-D is to provide a scalable architecture for executing incremental queries over large models. Our approach is based on the following foundations: (i) a distributed model storage system that (ii) supports a graph-oriented data representation format, and (iii) a graph query language adapted from the EMF-INCQUERY framework. The novel contribution of this report is an architecture and an implementation that consists of a (i) distributed model management middleware, and a (ii) distributed and stateful pattern matcher network based on the Rete algorithm.

INCQUERY-D provides incremental query execution by *indexing model contents* and *capturing model manipulation operations* in the middleware layer, and *propagating change tokens* along the pattern matcher network to *produce query results and query result changes* (corresponding to model manipulation transactions) efficiently. As the primary sources of memory consumption, i.e. both the indexing and intermediate Rete nodes can be distributed in a cloud infrastructure, the system is expected to scale well beyond the limitations of the traditional single workstation setup.

3.1 Case study: TrainBenchmark

Due to both confidentiality and technical reasons, it's difficult to obtain real-world industrial models and queries. Also, using confidential data sets hamstrings the reproducability of the conducted benchmarks. Therefore, we used an artificial data set which mimics real-world models.

In the following section we present the *TrainBenchmark*, a benchmark scheme and environment. The TrainBenchmark was designed and implemented by Benedek Izsó, István Ráth and Zoltán Szatmári [35]. The original goal of the TrainBenchmark was to compare various (preferably incremental) query engines' performance to EMF-INCQUERY's. For INCQUERY-D, we used a slightly modified version of the Train-

Benchmark.

3.1.1 Domain

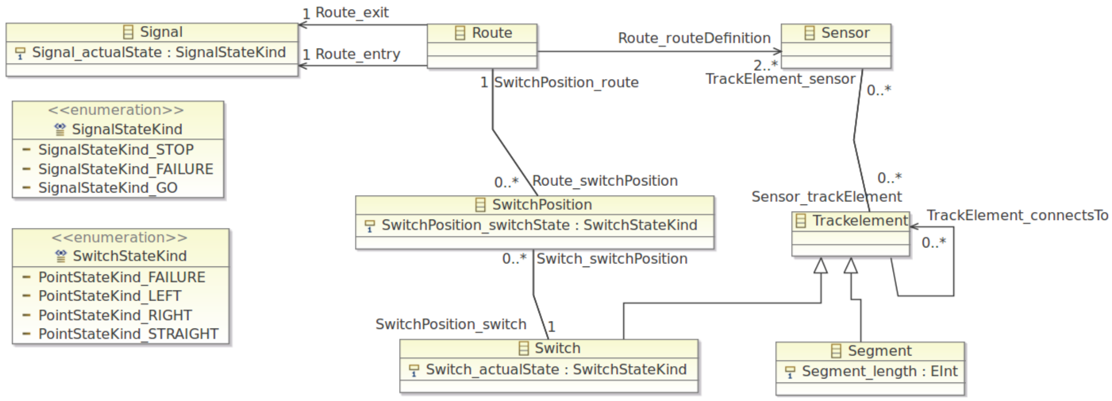

Figure 3.1: *The EMF metamodel of the TrainBenchmark*

The TrainBenchmark models is an imaginary railroad network. The network is composed of typical railroad items, including signals, segments, switches and sensors. The complete EMF metamodel of the TrainBenchmark is shown on Figure 3.1.

The *generator* project of TrainBenchmark is capable of generating railroad instance models of different sizes. It's capable of generating models in different formats, including EMF, OWL, RDF and SQL.

For Neo4j (subsection 2.3.3) and Titan (subsection 2.4.1), we expanded the generator with a module that can generate property graphs based on the TrainBenchmark's metamodel. It supports the GraphML [18], the Blueprints GraphSON [10] and the Faunus GraphSON [9] output formats.

3.1.2 Queries

The TrainBenchmark consists of queries that resemble a typical MDE workload. In general, MDE queries are more complex than those used in traditional databases. They often define large patterns with multiple join operations. The TrainBenchmark's queries look for violations of well-formedness constraints in the model. Although the TrainBenchmark defines four different queries, in this report, we only discuss the *RouteSensor* query in detail.

RouteSensor

The *RouteSensor* query looks for *Sensor*s that are connected to a *Switch*, but the sensor and the switch are *not* connected to the same *Route*. The graphical representation of

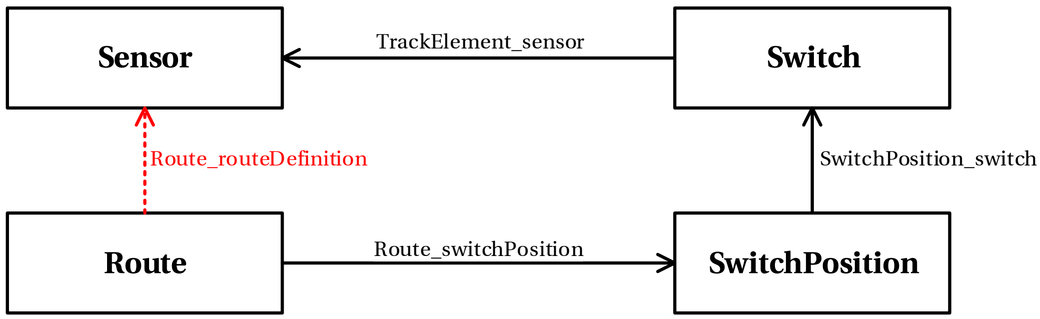

Figure 3.2: *Graphical representation of the* RouteSensor *query's pattern. The dashed red arrow defines a negative condition.*

the RouteSensor query is shown on Figure 3.2. The RouteSensor query IQPL (subsection 2.7.2).[1]

```
package hu.bme.mit.train.constraintcheck.incquery

import "http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl"

pattern routeSensor(Sen, Sw, Sp, R) = {
  Route(R);
  SwitchPosition(Sp);
  Switch(Sw);
  Sensor(Sen);

  Route.Route_switchPosition(R, Sp);
  SwitchPosition.SwitchPosition_switch(Sp, Sw);
  Trackelement.TrackElement_sensor(Sw, Sen);

  neg find head(Sen, R);
}

pattern head(Sen, R) = {
  Route.Route_routeDefinition(R, Sen);
}
```

Listing 3.1: *The* RouteSensor *query in IQPL*

```
PREFIX base: <http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT DISTINCT ?xSensor
WHERE
{
    ?xRoute rdf:type base:Route .
    ?xSwitchPosition rdf:type base:SwitchPosition .
    ?xSwitch rdf:type base:Switch .
    ?xSensor rdf:type base:Sensor .
```

[1] Note that the two queries are slightly different: the SPARQL query returns only a set of sensors, while the IQPL query returns a set of (Sensor, Switch, SwitchPosition, Route) tuples. TODO explain.

```
    ?xRoute base:Route_switchPosition ?xSwitchPosition .
    ?xSwitchPosition base:SwitchPosition_switch ?xSwitch .
    ?xSwitch base:TrackElement_sensor ?xSensor .

    FILTER NOT EXISTS {
        ?xRoute ?Route_routeDefinition ?xSensor .
    } .
}
```

Listing 3.2: *The* RouteSensor *query in SPARQL*

3.2 Architecture overview

In the following section, we provide an overview of the Rete algorithm, which forms the theoretical basis of EMF-INCQUERY and INCQUERY-D. We also describe INCQUERY-D's architecture.

3.2.1 Rete in general

INCQUERY-D's is based on the Rete algorithm, which provides incremental graph pattern matching. Originally created by Charles Forgy [28] for expert systems. Gábor Bergmann adapted it for EMF models and added many tweaks and improvements to the algorithm [22].

The Rete algorithm defines an asynchronous network of communicating nodes. This is essentially a dataflow network, with two types of nodes. Change notification objects (*tokens*) are propagated to intermediate *worker nodes* that perform operations (like filtering tokens based on constant expressions, or performing join or antijoin operations based on their contents) and store partial (interim) query results in their own memory. In contrast, *production nodes* are terminators that provide an interface for fetching query results and also their changes. Connections between nodes can be *local* (within one host) or *remote* (when two Rete nodes are allocated to different hosts).

It is important to emphasize that the database shards and Rete nodes are two distinct levels of distribution that do not directly depend upon each other.

3.2.2 INCQUERY-D architecture

INCQUERY-D's architecture consists of three layers: the storage layer, the middleware and the production network. The *storage layer* is a distributed database which is responsible for persisting the graph. The client application communicates with the *middleware*. The middleware provides a unified API for accessing the database. It also sends change notifications to the production network and retrieves the query results from the production network. The *production network* is implemented with a distributed Rete net which provides incremental query evaluation.

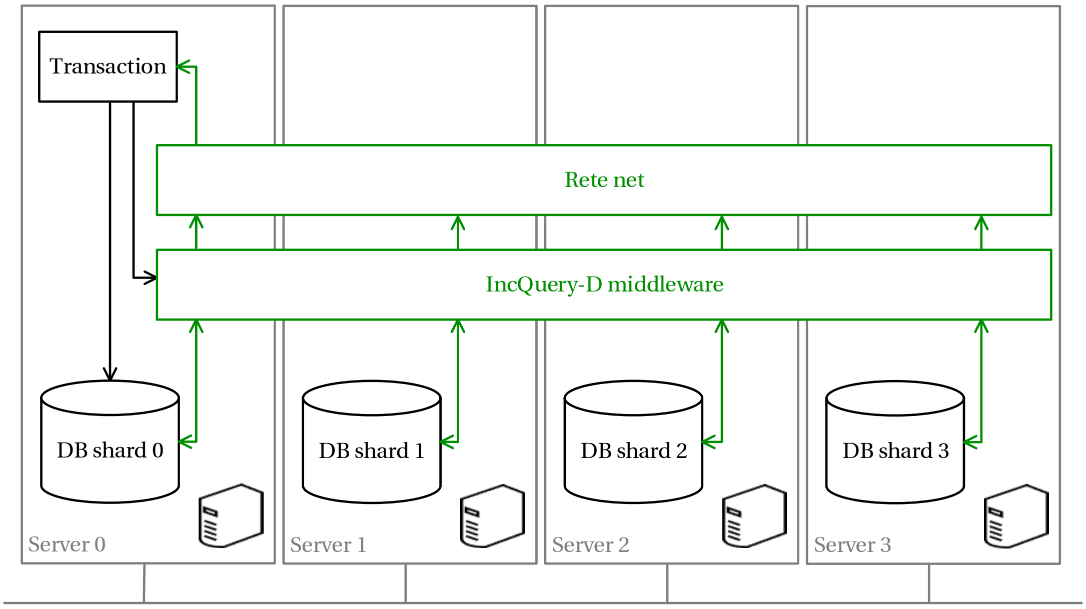

Figure 3.3: INCQUERY-D*'s architecture demonstrated on a four-node cluster*

The INCQUERY-D architecture in an example configuration scenario is shown in Figure 3.3.

3.3 Initialization and indexing

3.3.1 Indexing

Indexing is a common technique for decreasing the execution time of database queries. In MDE, *model indexing* is the key to high performance model queries. As MDE primarily uses a metamodeling infrastructure, the INCQUERY-D middleware maintains type-instance indexes so that all instances of a given type (both edges and graph nodes) can be enumerated quickly. These indexers form that bottom layer of the Rete production network.

3.3.2 Graph-like data manipulation

INCQUERY-D's middleware exposes an API that provides methods to manipulate the graph. By allowing graph-like data manipulation we allow the user to focus on the domain-specific challenges, thus increasing her productivity. The middleware translates the user's operation and forwards it to the underlying data storage (e.g. SPARQL queries for 4store and Gremlin queries for Titan).

Data representation

Conceptually, the architecture of INCQUERY-D allows the usage of a wide scale of model representation formats. Our prototype has been evaluated in the context of the *property graph* and the *RDF* data model, but other mainstream metamodeling and knowledge representation languages such as relational databases' SQL dumps and Ecore [45] could be supported, as long as they can be mapped to an efficient and distributed storage backend (e.g. triplestores, key-value stores or column-family databases).

To support different data models, we only have to supply the appropriate connector classes to INCQUERY-D's middleware. The current implementation supports 4store and Titan.

3.3.3 Notification mechanisms

Model change notifications are required by incremental query evaluation, thus model changes are captured and their effects propagated in the form of *notification objects* (NOs). To notifications are responsible for maintaining the Rete network's state. INCQUERY-D's middleware layer achieves this by providing a facade for model manipulation operations.

Current database managment systems

While relational databases usually provide *trigger*s for generating notifications, most triplestores and graph databases lack this feature. Among our primary database backends, 4store provides no triggers at all. Titan and Neo4j incorporate Blueprints, which provides an `EventGraph` class, which is capable of generating notification events, but only in a single JVM. Implementing distributed notifications would require us to extend the `EventGraph` class and using a messaging framework (see Future work XXX).

Because of these reasons, in INCQUERY-D's current implementation, notifications are controlled by the middleware. The notification messages are propagated through the Rete network via the Akka messaging framework.

3.4 Incremental queries, change propagation

3.4.1 Rete

Detailed Rete with an actual instance model

Similar algorithms

TREAT, LEAPS, newer Rete variants

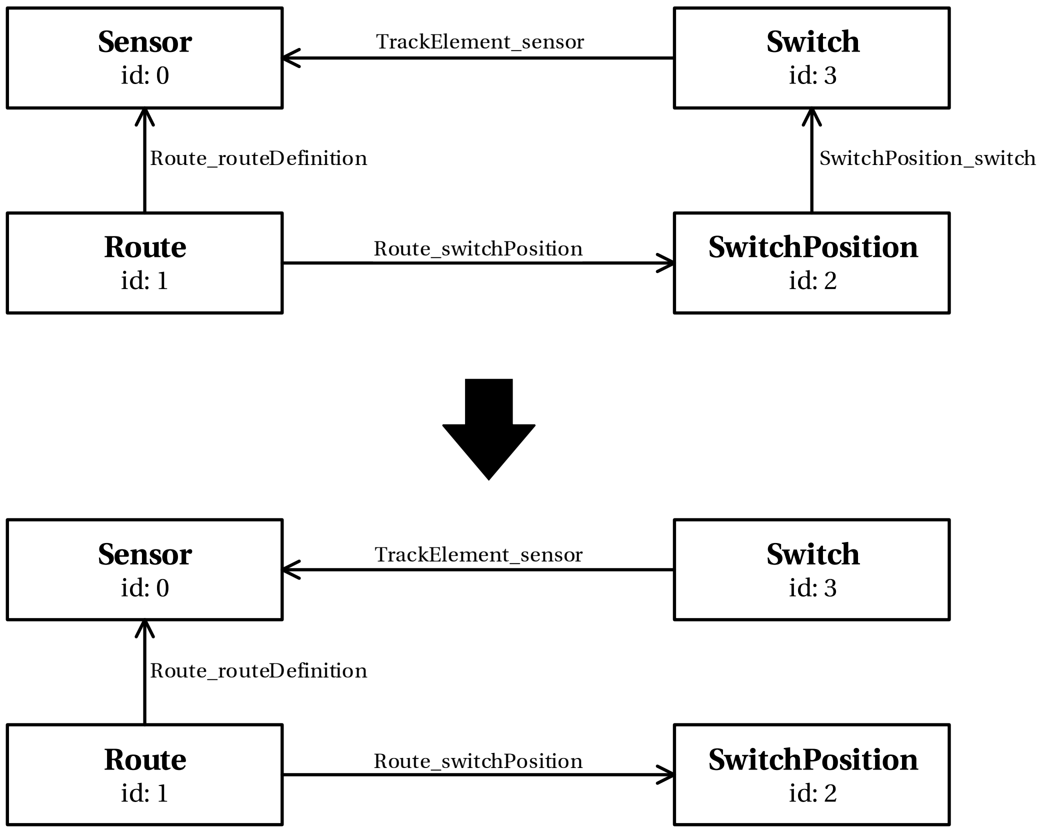

Figure 3.4: *A modification on a* TrainBenchmark *instance model*

TREAT [37]

LEAPS [21]

3.4.2 Distributed operation

Principles

Practice

??

Transparent framework: Akka

3.4.3 Scalability considerations

For the storage layer, the most important issue from an incremental query evaluation perspective is that the indexers of the middleware should be filled as quickly as possible. This favors technologies where model sharding can be performed efficiently (i.e. with balanced shards in terms of type-instance relationships), and elementary queries (or model graph traversals) can be executed efficiently.

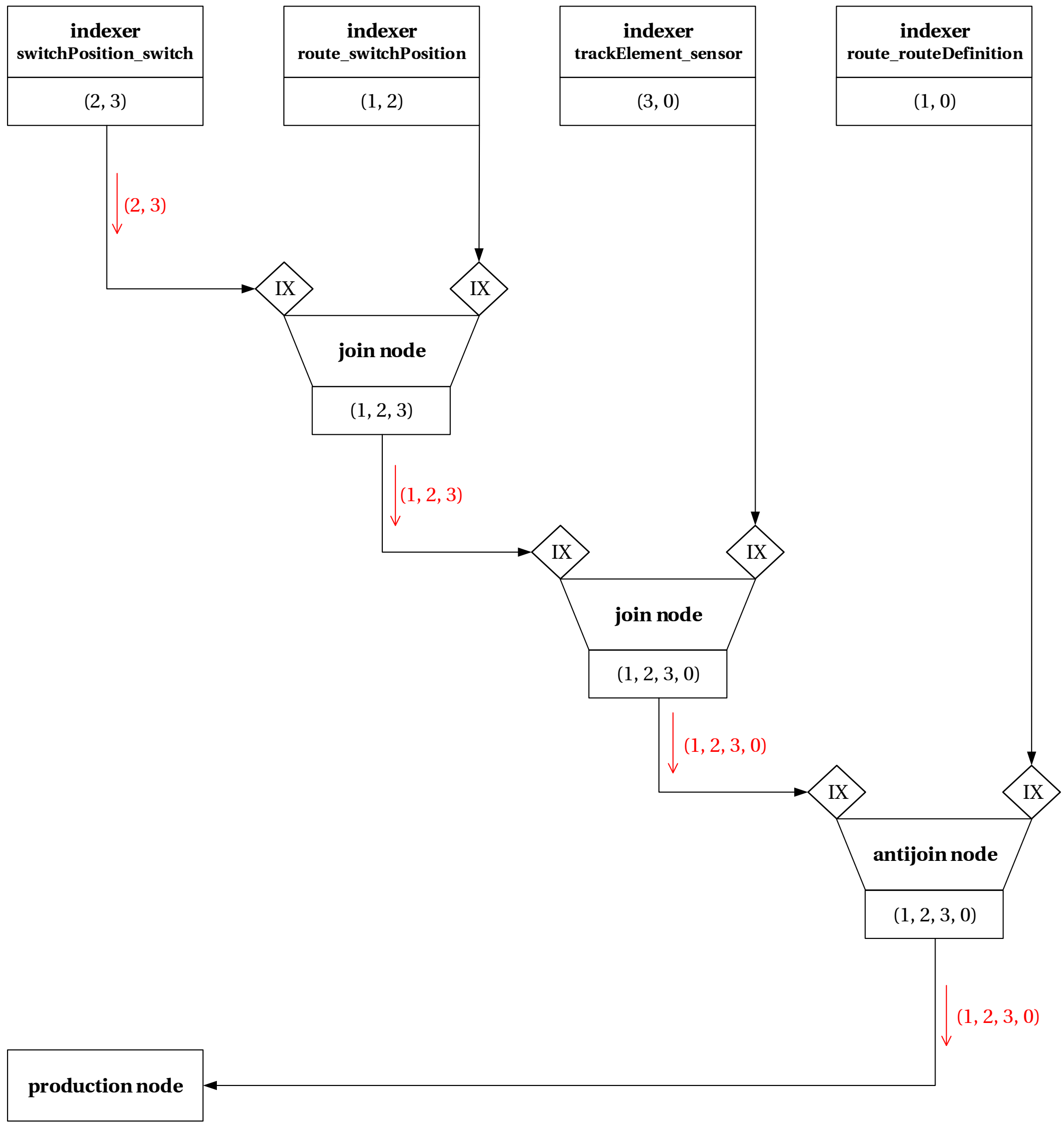

Figure 3.5: *The Rete net and the partial matches stored in its nodes*

Achieving scalability of the distributed Rete architecture is an equally complex challenge. The overall performance of the system is influenced by a number of factors, including (i) the *layout of the Rete network* (which can be optimized depending on both query and instance model characteristics, e.g. to keep the resource requirement of intermediate join operations to a minimum), (ii) the *allocation* of Rete nodes to host computers (e.g. to optimize local resource usage, or to minimize the amount of remote network communication), and (iii) *dynamic adaptability* to changing conditions (e.g. when the model size and thus query result size grows rapidly, the Rete network may require dynamic reallocation or node sharding due to local resource limitations).

3.5 Deployment, configuration

Deploying, configuring and operating a distributed pattern matcher is a complex task. In the following, we will present our tools for these tasks.

3.5.1 Tooling

reuse of EMF-INCQUERY's components
Eclipse-based tree editor and yWorks viewer

3.5.2 Degrees of freedom

database sharding, allocation of rete nodes – orthogonal
choosing different dbs

Different database implementations

diffent storage backends are supported
storage agnostic representation

3.5.3 Workflow

In the following part, we will describe the workflow behind the pattern matching process. Starting from a metamodel, an instance model and a graph pattern, we will cover the problem pieces that need to be solved for setting up an incremental, distributed pattern matcher. The workflow is shown on Figure 3.6.

Analyze the metamodel and the query

Task. First, we determine the constraints defined by the query pattern. The matches satisfying these constraints will define the results of the query.

Implementation. The pattern is defined in an IQPL (INCQUERY Pattern Language) text file. Using Xtext [19], an Eclipse-based framework for creating domain-specific languages, the textual representation of the pattern is parsed to an EMF model. Based on the EMF model's pattern and the metamodel, a constraint network called PSystem (short for *Pattern System*) is generated.

Build a Rete layout

Task. To allow incremental query evaluation, we create a Rete net based on the constraints derived from the query.

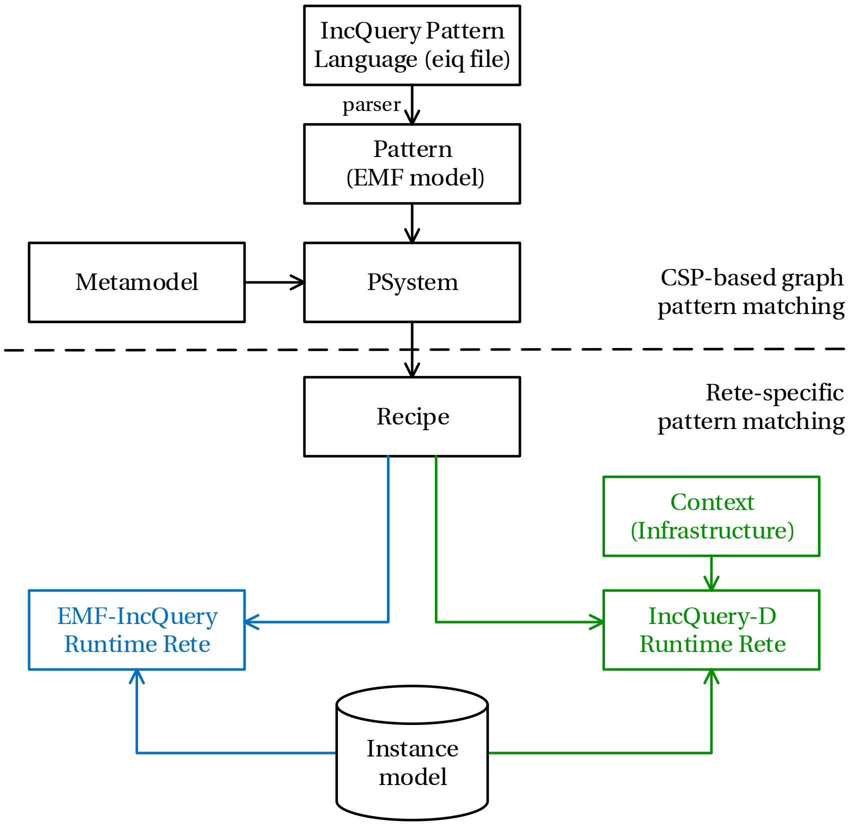

Figure 3.6: *The workflow of* EMF-INCQUERY *(blue) and* INCQUERY-D *(green)*

Implementation. As we mentioned earlier, we aim to reuse as much of EMF-INCQUERY's existing code base as possible. As part of this attempt, we introduced the concept of *Rete recipe*s which define the layout of a Rete network.

Allocate the Rete network in the cloud's nodes

Task. Because of its single workstation nature, EMF-INCQUERY simply unfolds the Rete net based on the derived Rete recipe. At the same time, INCQUERY-D operates in a distributed environment where local resource exhaustion, network latency and throughput are critical aspects.

Implementation. Currently, the allocation of the Rete nodes is done manually. To address this limitation, we plan to utilize Constraint Satisfaction Problem (CSP) solvers, or dynamic techniques like Design Space Exploration (DSE) [33].

Bootstrap the system

Task. Based on the Rete network's allocation, we have to deply the Rete nodes in the distributed systems. After the successful deployment, the Rete network has to be initialised. Due to the Rete algorithm's asynchronous nature, it uses a termination protocol to signal when the data processing is finished.

Implementation. In INCQUERY-D's prototype, both the bootstrapping and the Rete network's operation is carried out automatically. The Akka actors representing the Rete nodes are deployed and initiated using Akka's *programmatic remote deployment* feature.

3.6 Elaboration of the example

We use the *RouteSensor* query as our example. The query is shown as a graph pattern definition on Listing 3.1 and visualized on Figure 3.2. Queries like this are typical in MDE applications (such as well-formedness validation or complex model transformations).

3.6.1 Workflow

Following the workflow defined in subsection 3.5.3, we will cover the actual steps for deploying and operating a distributed pattern matcher for the *RouteSensor* query.

Analyze the metamodel and the query

The metamodel is shown on Figure 3.1. Using EMF-INCQUERY's tooling, the textual representation (`routeSensor.arch`, see Listing 3.1) is analyzed and parsed to an EMF model (Figure 3.7).

Build a Rete layout

Based on the query's EMF model, EMF-INCQUERY's tooling builds PSystem and creates a Rete layout, that quarantees the satisfaction of the constraints. The Rete layout is shown on Figure 3.5.

Allocate the Rete network in the cloud's nodes

In INCQUERY-D's current implementation, the Rete recipe's nodes are allocated manually on the cloud servers (called *Machine*s). The allocation is currently defined in an architecture file (e.g. `routeSensor.arch`). The Rete nodes are associated with the machines with *infrastructure mapping* edges.

INCQUERY-D's tooling currently provides an Eclipse-based tree editor to define machines and the infrastructure mapping edges (Figure 3.8).

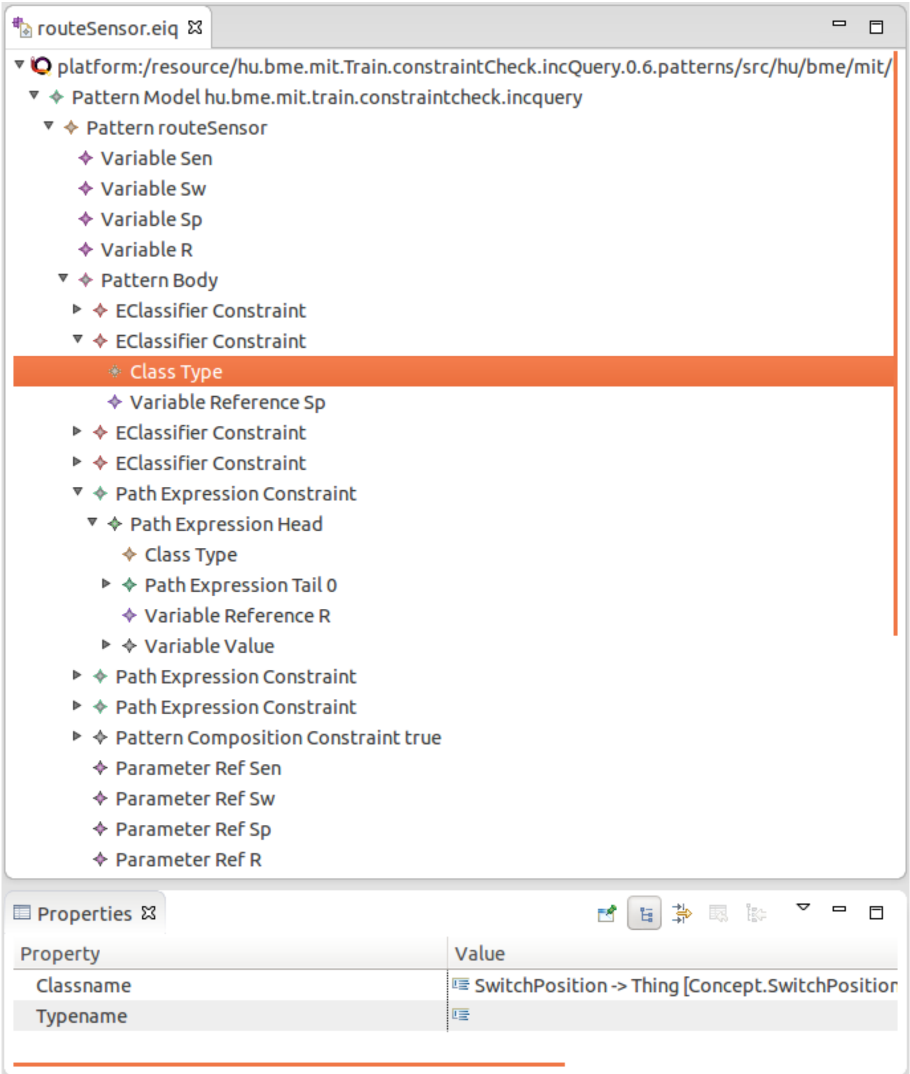

Figure 3.7: *The EMF model generated from the pattern*

The tooling is capable of visualizing the Rete network and its mapping to the machines (see Figure 3.9)

Bootstrap the system

INCQUERY-D's current implementation, the distributed system is initiated with a Bash script which launches the Akka microkernel on the appropriate nodes. The Akka actors representing the Rete network's nodes are deployed automatically by the INCQUERY-D *Coordinator* node.

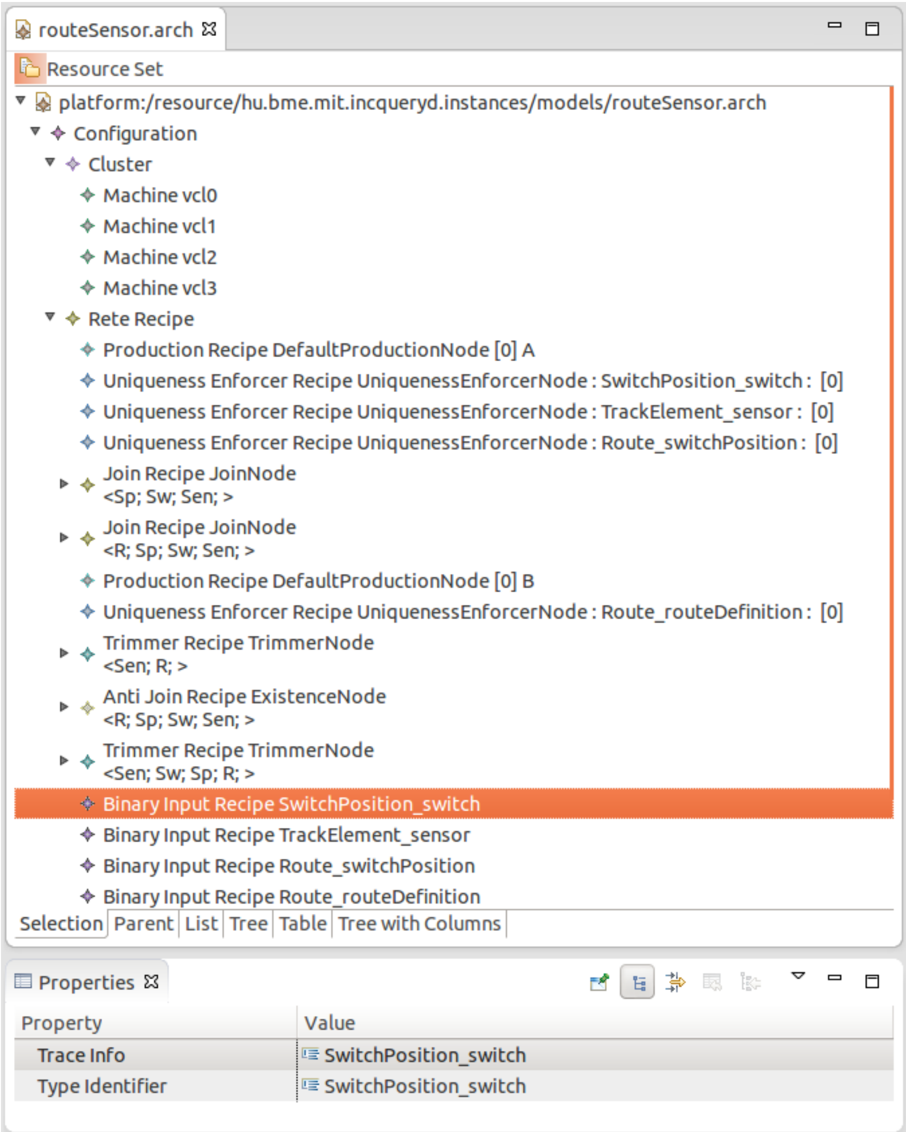

Figure 3.8: *The tree editor in* INCQUERY-D*'s tooling*

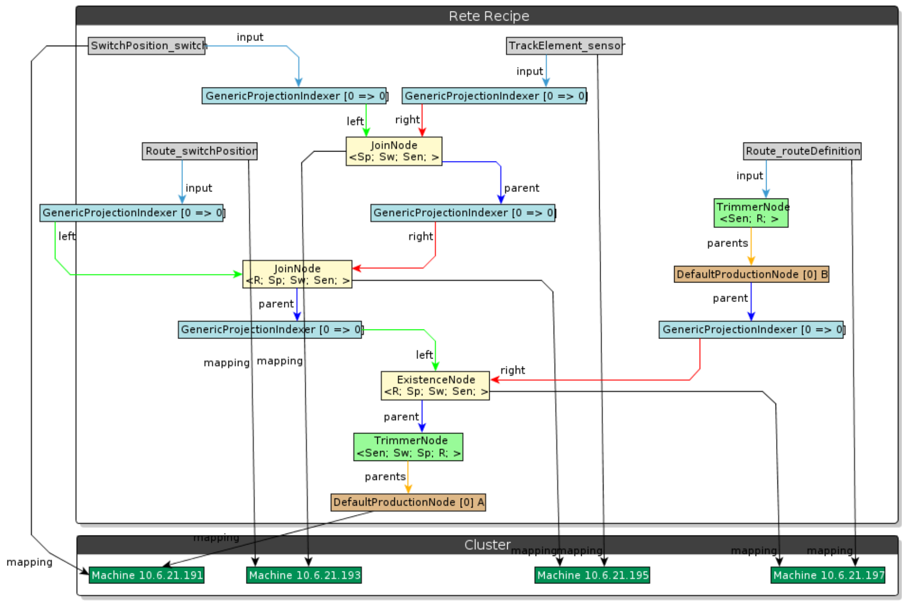

Figure 3.9: *The yFiles viewer in* INCQUERY-D*'s tooling*

Chapter 4

Evaluation

4.1 Goals

4.1.1 Phases

The TrainBenchmark consists of the following phases:

1. *read*: loading the model,
2. $check_0$: running the queries,
3. $edit_i$: editing the model,
4. $check_i$: running the queries again.

In a ”real-world” model editing sequence, the user tipically edits the model in small steps ($edit_i$ phases). The user's work is much more productive if she receives an instant feedback, hence we would like to run re-evaluate well-formedness queries quickly (preferably in sub-second time). This creates the need for an incremental pattern matcher tools.

4.1.2 Measure the response time and the scalability

4.1.3 Workload profile's difference from standard benchmarks

4.2 Benchmark scenario

In order to measure the efficiency of model queries and manipulation operations over the distributed architecture, we designed a benchmark to measure tool response times in a well-formedness validation use case. The benchmark transaction sequence consists of four phases: (i) during the *load* phase, the serialization of the model is loaded into the database; (ii) a test query (**??**) is executed ($check_0$); finally, in a cycle consisting of several repetitions, some elements are programmatically modified ($edit_i$) and the test query is re-evaluated ($check_i$). We ran the benchmark on pseudo-randomly generated instance models of growing size, each model containing about twice as many elements (vertices and edges) as the previous one and having a regular fan-out structure. As the current version of Neo4j does not have built-in support for graph sharding, the benchmark uses a manually sharded strategy where each shard contains a disjoint partition of the model.

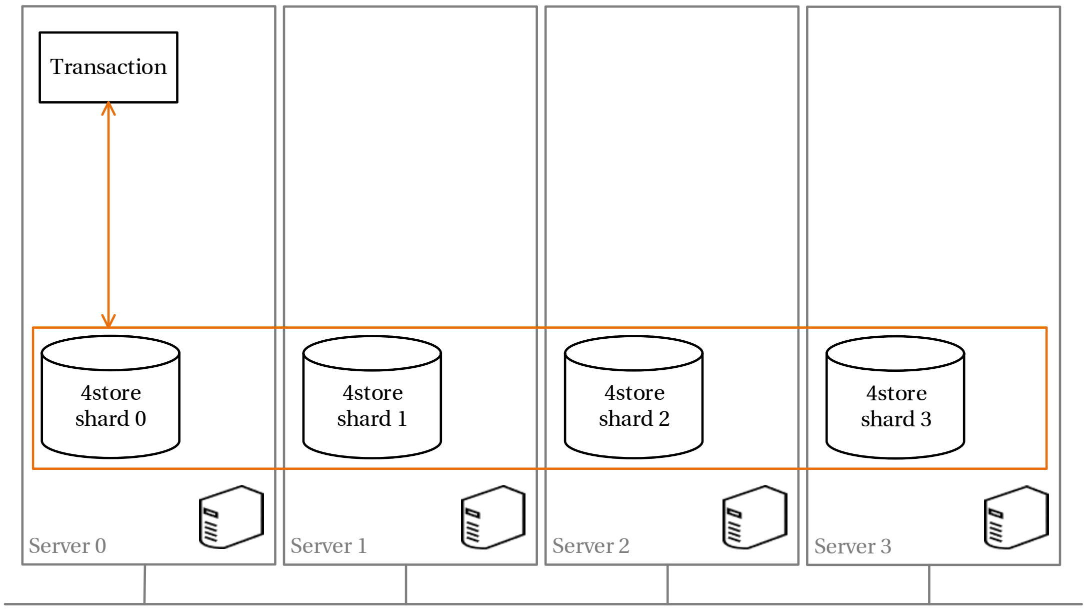

Figure 4.1: *The non-incremental baseline benchmark's setup*

4.3 Generation of models

We created a property graph generator project based on the previous TrainBenchmark generators. The generator creates a graph in an embedded Neo4j database and uses the Blueprints library's `GraphMLWriter` class to save to a GraphML file [8].

We implemented INCQUERY-D as an initial prototype to evaluate the feasibility of the approach, and to experiment with various optimization possibilities. As the storage, we used the popular graph database Neo4j [39] featuring automatic indexes and two core query technologies (Gremlin and Cypher) that were used as a low-level model access interface by our middleware layer. The prototype of the distributed middleware and Rete network were implemented in Java using Akka [5], the Scala-based toolkit for building applications based on the Actor model, since it is well-suited for asynchronous applications. The communication protocol was built on top of Akka's built-in serialization support.

4.4 Benchmark environment

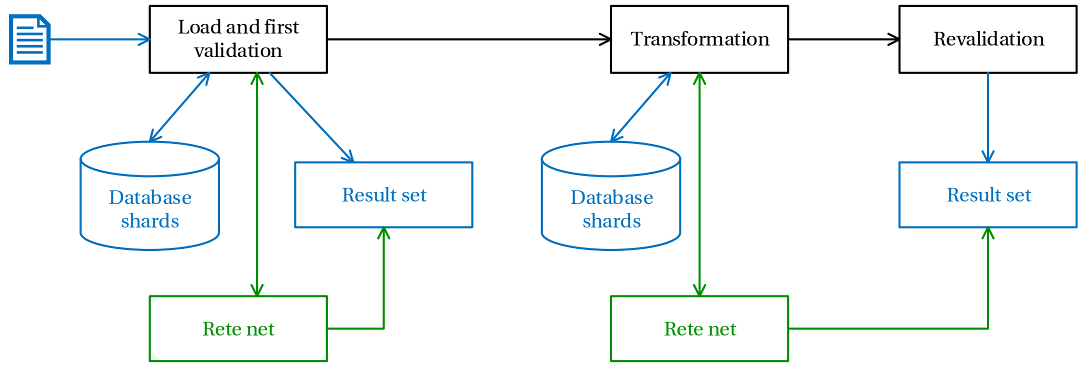

Figure 4.2: *The benchmark scenario*

4.4.1 Benchmark setup

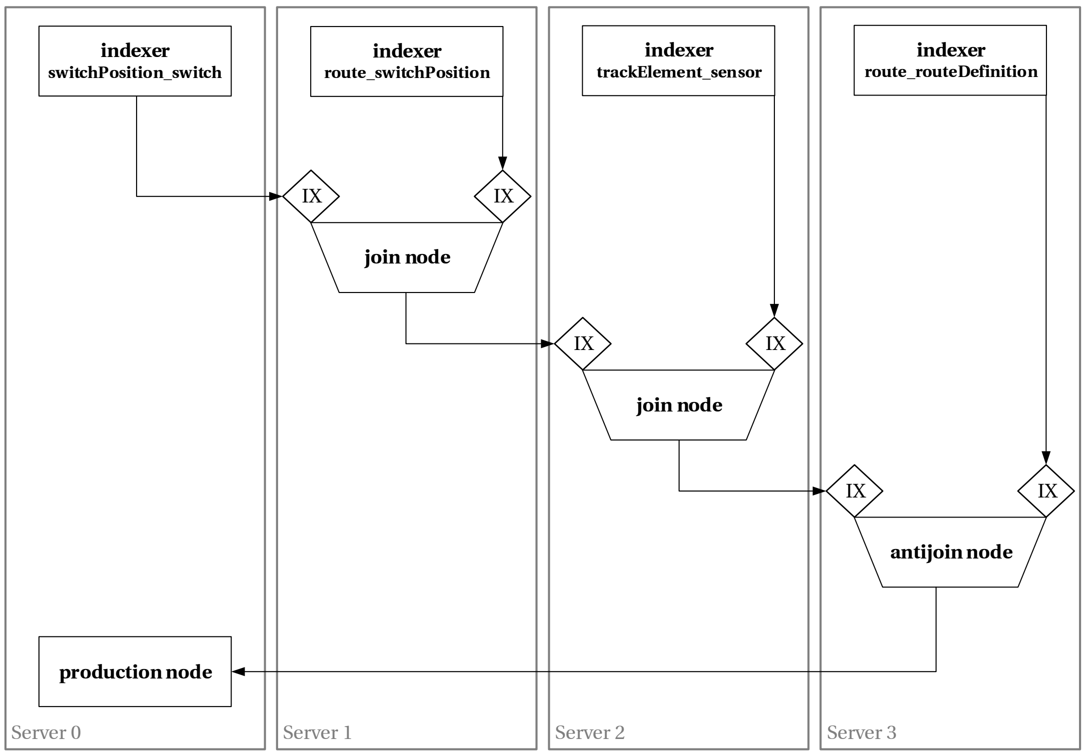

Figure 4.3: *The layout of the distributed Rete net*

4.4.2 Hardware, software ecosystem

As the testbed, we deployed our system to a private cloud consisting of 4 virtual machines on separate host computers.

Hardware

For Titan and 4store, each virtual machine used dual 2.5 GHz Intel Xeon L5420 CPUs with 8 GBs of RAM, running on Ubuntu 12.10 64-bit.

For Neo4j, each virtual machine had the same parameters but twice as much, 16 GBs of RAM.

Software

We used the following technologies:

- Ubuntu 12.10 64-bit
- Oracle Java 7 64-bit
- 4store 1.1.5
- Titan 0.3.2
- Faunus 0.3.2
- Hadoop 1.1.2
- Cassandra 1.2.2
- Neo4j 1.8
- Blueprint 2.3.0
- Akka 2.1.2
- Eclipse 4.3 (Kepler)

4.4.3 Benchmark methodology

4.4.4 Data collection

4.4.5 Data processing tools

To compare the performance characteristics of INCQUERY-D to a traditional case, we defined two scenarios. The *batch* scenario uses only Neo4j to manage models and evaluate the queries in a parallelized way (depicted as ① in **??**). This serves as a baseline for the *incremental* scenario, which uses INCQUERY-D (shown as ② in **??**). For these initial experiments, the layout and allocation of the Rete network was determined manually.

4.5 Results

The measurement results of our experiments are shown in **??** (aggregated from several complete sets to filter transient effects). As expected, the *load* phase take about the same time for both scenarios, and INCQUERY-D is about half an order of magnitude slower when evaluating the query at first ($check_0$ phase) due to the Rete construction overhead. However, INCQUERY-D is several orders of magnitude faster during the $edit_i - check_i$ cycles, making on-the-fly query (re)evaluation feasible even for models larger than 50 million elements. Once initialized, INCQUERY-D scales linearly, since query response times for growing models can be kept low by adding additional computers for hosting Rete nodes.

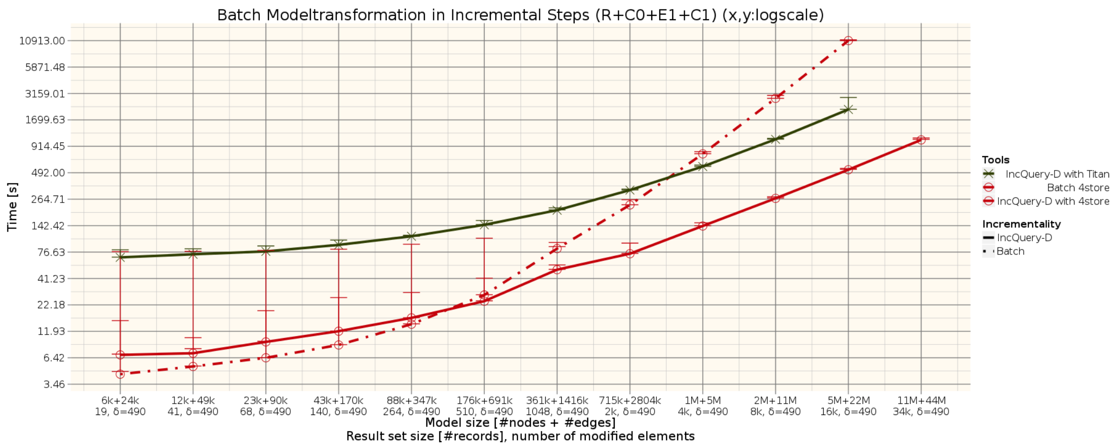

Figure 4.4: *Batch transformation*

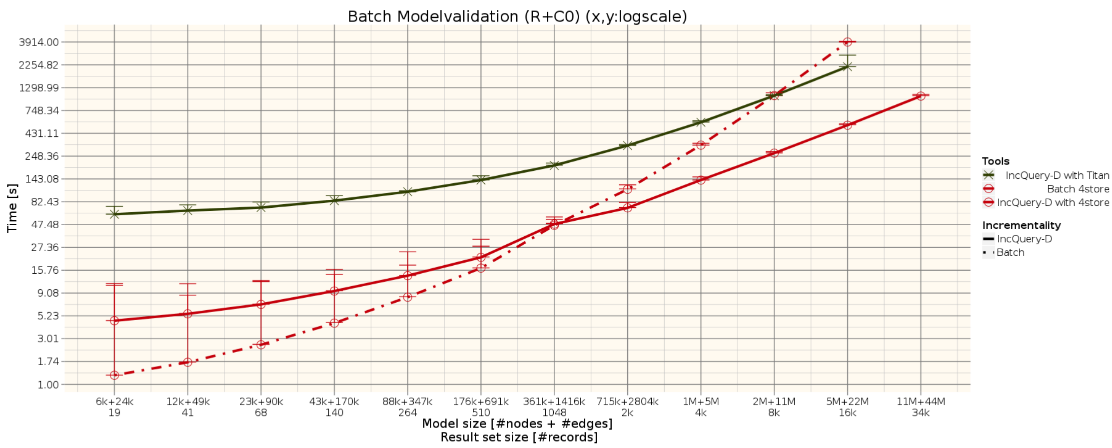

Figure 4.5: *Batch validation*

4.5.1 Results using the Neo4j graph database

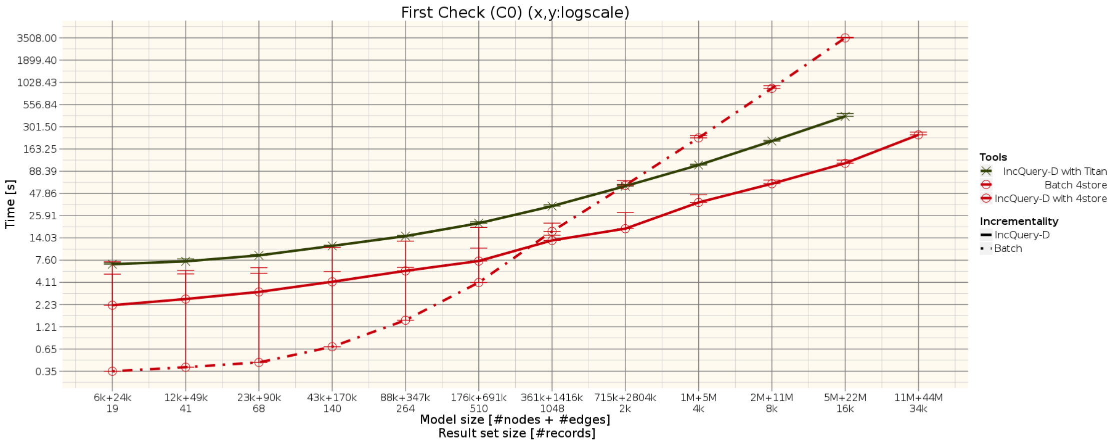

Figure 4.6: *check$_0$ phase*

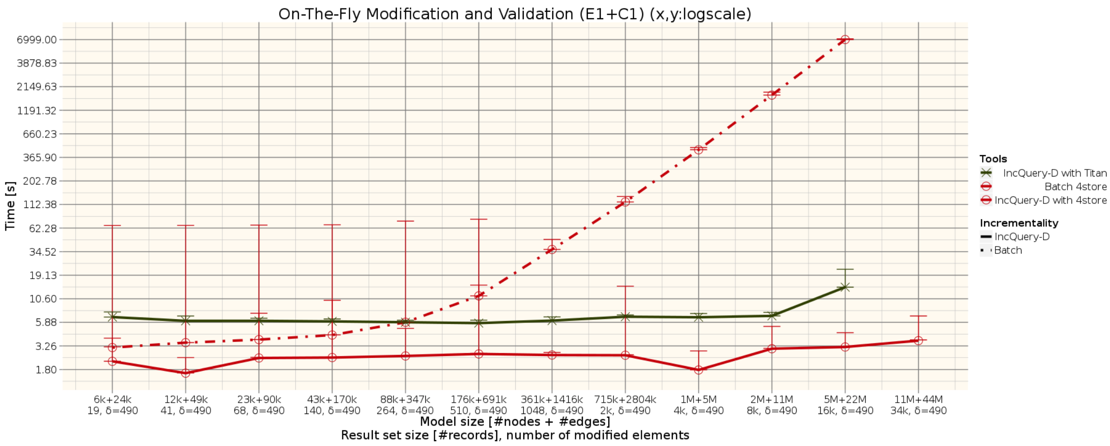

Figure 4.7: *On-the-fly revalidation (edit and check$_1$ phase)*

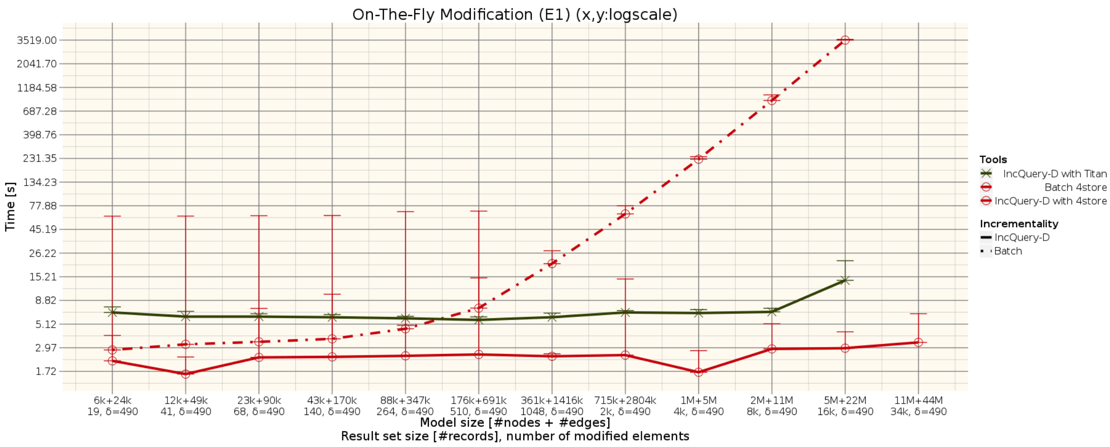

Figure 4.8: *On-the-fly revalidation, edit phase*

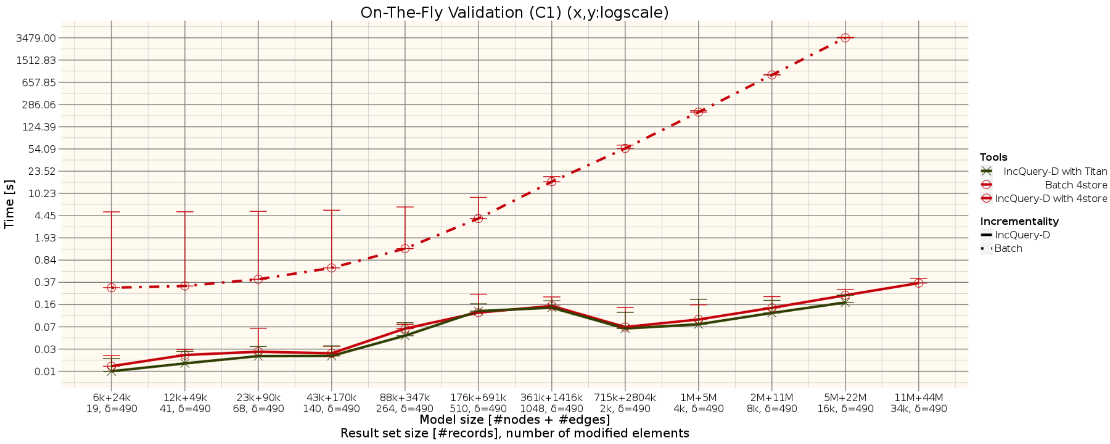

Figure 4.9: *On-the-fly revalidation, check$_1$ phase*

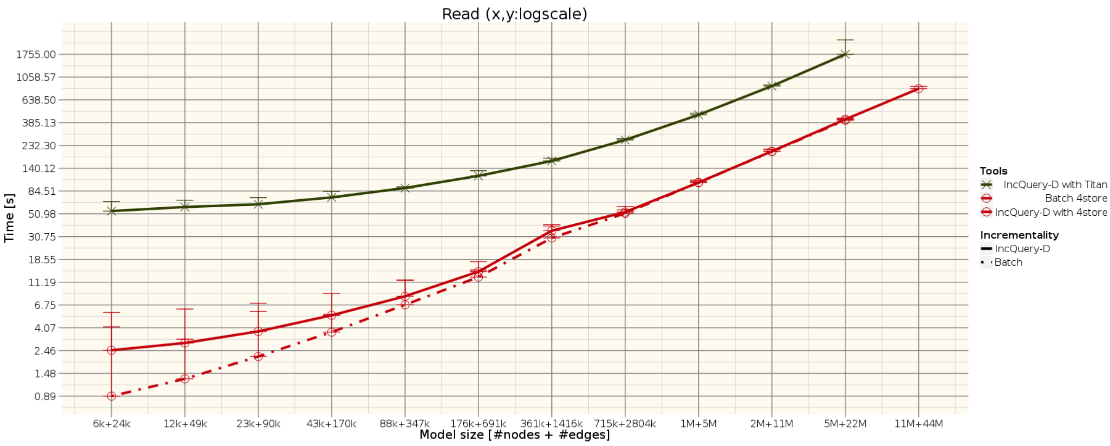

Figure 4.10: *Read phase*

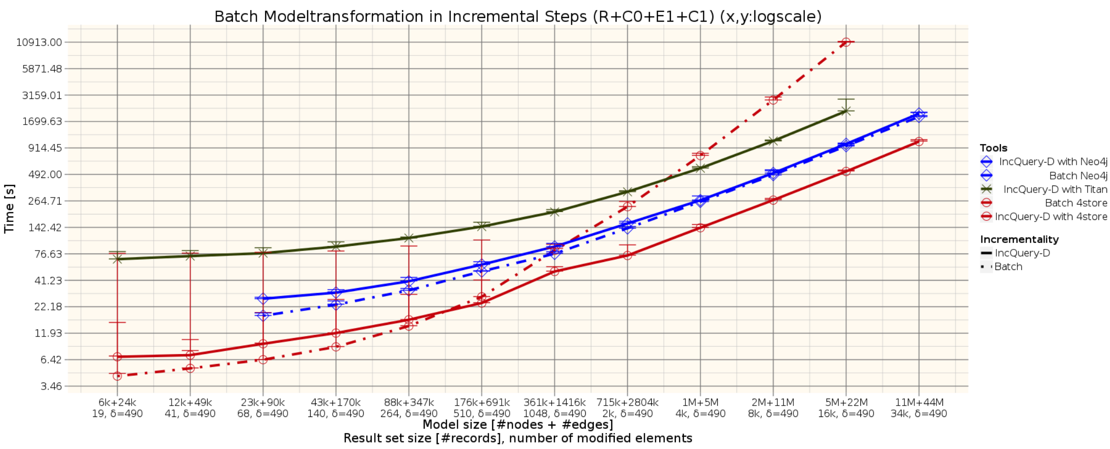

Figure 4.11: *Batch transformation*

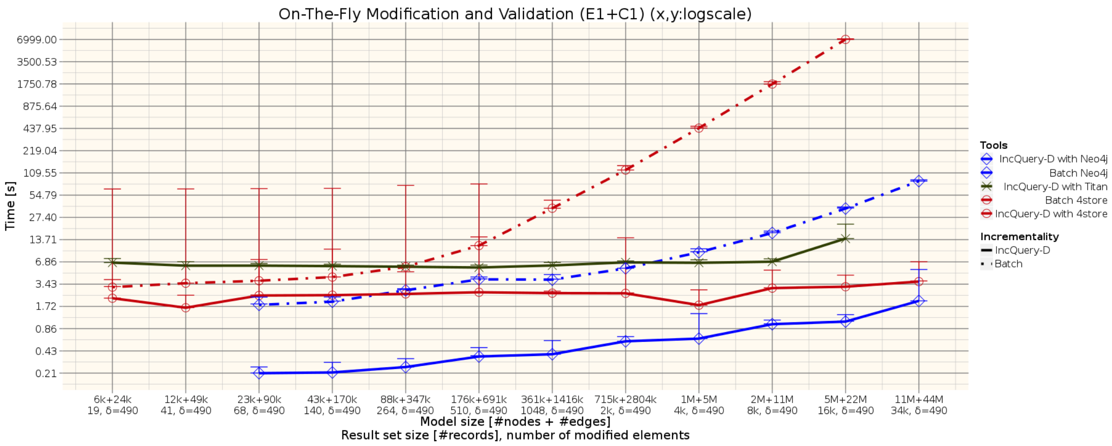

Figure 4.12: *On-the-fly revalidation (edit and check$_1$ phase)*

Chapter 5

Related and future work

5.1 Related work

A wide range of special languages have been developed to support *graph based* representation and querying of computer data. A class-diagram like modeling language is Ecore of the Eclipse Modeling Framework (EMF [45]), where classes, references between them and attributes of classes describe the domain. Extensive tooling helps the creation and transformation of such domain models. For EMF models, OCL is a declarative constraint description and query language that can be evaluated with the local-search based Eclipse OCL [27] engine. To address scalability issues, impact analysis tools [30] have been developed as extensions or alternatives to Eclipse OCL.

Outside the Eclipse ecosystem, the Resource Description Framework (RDF [31]) is developed to support the description of instances of the semantic web, assuming sparse, ever-growing, incomplete data. Semantic models are built up from triple statements and they can be queried using the SPARQL [46] graph pattern language with tools like Sesame [2] or Virtuoso [1]. Property graphs [41] provide a more general way to describe graphs by annotating vertices and edges with key-value properties. Such data structures can be stored in graph databases like Neo4j [39] which provides the Cypher [44] query language. Even though big data storage (usually based on MapReduce) provides fast object persistence and retrieval, query engines realized directly on these data structures do not provide dedicated support for incremental query evaluation.

In the context of event-based systems, distributed evaluation engines were proposed earlier [20]. However they scaled up in the number of rules [24] rather than in the number of data elements. As a very recent development, Rete-based caching approaches have been proposed for the processing of Linked Data (bearing the closest similarity of our approach). INSTANS [40] uses this algorithm to perform complex event processing (formulated in SPARQL) on RDF data, gathered from distributed sensors. Diamond [38] evaluates SPARQL queries on Linked Data, but it lacks an indexing middleware layer so their main challenge is efficient data traversal.

The conceptual foundations of our approach as based on EMF-INCQUERY [23], a tool that evaluates graph patterns over EMF models using Rete. Up to our best knowledge, INCQUERY-D is the first approach to promote distributed scalability by *distributed incremental query evaluation* in the MDE context. As the architecture of INCQUERY-D separates the data store from the query engine, we believe that the scalable processing of property graphs can open up interesting applications outside of the MDE world.

Acharya et al. described a Rete network mapping for fine grained and medium grained message-passing computers [20]. The medium-grained computer connected processors in a crossbar architecture, while our approach use computers connected by gigabit Ethernet. The paper published benchmark results of the medium-grained solution, but these are based only on simulations.

5.2 Future work

For future work, we plan on providing more sophisticated automation for sharded Ecore models, and further exploring advanced optimization challenges such as dynamic reconfiguration and fault tolerance. We also plan experiment with programming languages that are better suited to asynchronous algorithms (e.g. Erlang and Scala) and try different database systems (e.g. MongoDB) as our storage layer.

Chapter 6

Conclusions

6.1 Summary of contributions

In this chapter, ...

6.1.1 Own work

During the research and development of INCQUERY-D's prototype, I achieved the following results:

- I expanded the TrainBenchmark with a *new instance model generator*, which can produce property graphs and serialize them in various formats: GraphML, Blueprints GraphSON and Faunus GraphSON.
- I implemented a *distributed, asynchronous version of the Rete algorithm* with a termination protocol. Based on the Rete algorithm, I created a *distributed incremental query engine's prototype*, which is not only detached from the data storage backend, but also agnostic to the storage backend's data model. To prove this, the query engine was tested with both property graphs and RDF graphs.
- Based on EMF-INCQUERY and István Ráth's work, I created an *Eclipse-based environment* to provide automated deployment of the Rete network. This allows the user to define complex queries in IQPL, a high-level pattern language.
- I experimented with contemporary non-relational database management systems with a focus on NoSQL graph databases and triple stores.
  - I implemented scripts to install the *Titan graph database and its ecosystem* on a cluster. Titan's ecosystem includes technologies on different maturity levels, including the Apache Cassandra database, the Apache Hadoop MapReduce framework with the HDFS distributed file system, the TinkerPop graph framework with the Gremlin query language and the Faunus graph analytics engine.
  - I implemented scripts to install the *4store triplestore* on a cluster. I created

the connector class in INCQUERY-D's middleware and formulated the necessary the SPARQL queries.
  - I deployed a manually sharded *Neo4j cluster.* I created the connector class in INCQUERY-D's middleware to use Neo4j's REST interface and formulated the appropriate Cypher queries.
- I implemented scripts for *automating the benchmark and operating a cluster of Akka microkernels.*
- I conducted a benchmark to measure INCQUERY-D's *response time and scalability characteristics.* For the benchmark's baseline, I created *distributed non-incremental benchmark scenarios,* with Neo4j and 4store.
- For INCQUERY-D's implementation, I wrote more than 3000 lines of Java code and approximately 500 lines of configuration and deployment scripts.

6.2 Limitations

- manual allocation of Rete nodes
- only a subset of Rete nodes implemented
- lack of complete tooling

6.3 Future directions

We presented INCQUERY-D, a novel approach to adapt distributed incremental query techniques to large and complex model driven software engineering scenarios. Our proposal is based on a distributed Rete network that is decoupled from sharded graph databases by a middleware layer, and its feasibility has been evaluated using a benchmarking scenario of on-the-fly well-formedness validation of software design models. The results are promising as they show nearly instantaneous query re-evaluation as model sizes grow well beyond 10^7 elements.

Bibliography

[1] OpenLink Software: Virtuoso Universal Server. http://virtuoso.openlinksw.com/.

[2] Sesame: RDF API and Query Engine. http://www.openrdf.org/.

[3] 2013: What's Coming Next in Neo4j! http://blog.neo4j.org/2013/01/2013-whats-coming-next-in-neo4j.html, January 2013.

[4] 4store website. http://4store.org/, October 2013.

[5] Akka website. http://akka.io/, May 2013.

[6] Apache Cassandra website. http://cassandra.apache.org/, May 2013.

[7] Apache Hadoop website. http://hadoop.apache.org/, May 2013.

[8] Blueprints website. http://blueprints.tinkerpop.com/, May 2013.

[9] GraphSON Format. https://github.com/thinkaurelius/faunus/wiki/GraphSON-Format, October 2013.

[10] GraphSON Reader and Writer Library. https://github.com/tinkerpop/blueprints/wiki/GraphSON-Reader-and-Writer-Library, October 2013.

[11] Guava: Google Core Libraries for Java 1.6+. https://code.google.com/p/guava-libraries/, May 2013.

[12] Lustre. http://lustre.org/, October 2013.

[13] Neoclipse website. https://github.com/neo4j/neoclipse, May 2013.

[14] NoSQL Databases. http://nosql-database.org/, May 2013.

[15] Objectivity – InfiniteGraph. http://www.objectivity.com/infinitegraph, October 2013.

[16] OrientDB Graph-Document NoSQL dbms. http://www.orientdb.org/, October 2013.

[17] Sparsity-technologies: DEX high-performance graph database. http://www.sparsity-technologies.com/dex, October 2013.

[18] The GraphML File Format. http://graphml.graphdrawing.org/, October 2013.

[19] Xtext – Language Development Made Easy! http://www.eclipse.org/Xtext/, October 2013.

[20] Acharya, A. et al. Implementation of production systems on message-passing computers. *IEEE Trans. Parallel Distr. Syst.*, 3(4):477–487, July 1992.

[21] Don Batory. The LEAPS Algorithm. Technical report, Austin, TX, USA, 1994.

[22] Gábor Bergmann. Incremental graph pattern matching and applications. Master's thesis, Budapest University of Technology and Economics, http://mit.bme.hu/~rath/pub/theses/diploma_bergmann.pdf, 2008.

[23] Bergmann, Gábor et al. Incremental evaluation of model queries over EMF models. In *MODELS*, volume 6394 of *LNCS*. Springer, 2010.

[24] Bin Cao, Jianwei Yin, Qi Zhang, and Yanming Ye. A MapReduce-Based Architecture for Rule Matching in Production System. In *CLOUDCOM*. IEEE, 2010.

[25] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[26] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[27] Eclipse MDT Project. Eclispe OCL website, 2011. http://eclipse.org/modeling/mdt/?project=ocl.

[28] Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligences*, 19(1):17–37, 1982.

[29] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[30] Thomas Goldschmidt and Axel Uhl. Efficient OCL impact analysis, 2011.

[31] RDF Core Working Group. Resource Description Framework (RDF). http://www.w3.org/RDF/, 2004.

[32] S. Harris, N. Lamb, and N. Shadbolt. 4store: The Design and Implementation of a Clustered RDF Store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, 2009.

[33] Ábel Hegedüs, Ákos Horváth, István Ráth, and Dániel Varró. A Model-driven Framework for Guided Design Space Exploration. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, Lawrence, Kansas, USA, 11/2011 2011. IEEE Computer Society, IEEE Computer Society. ACM Distinguished Paper Award, Acceptance rate: 15%.

[34] Guillaume Hillairet, Frédéric Bertrand, Jean Yves Lafaye, et al. Bridging EMF applications and RDF data sources. In *Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering, SWESE*, 2008.

[35] Benedek Izsó, Zoltán Szatmári, Gábor Bergmann, Ákos Horváth, István Ráth, and Varro Daniel. Ontology driven design of emf metamodels and well-formedness constraints. In *Proceedings of the 12th Workshop on OCL and Textual Modelling*, OCL '12, page 37–42, New York, NY, USA, 2012. ACM, ACM.

[36] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[37] D. P. Miranker and B. J. Lofaso. The Organization and Performance of a TREAT-Based Production System Compiler. *IEEE Trans. on Knowl. and Data Eng.*, 3(1):3–10, March 1991.

[38] Miranker, Daniel P et al. Diamond: A SPARQL query engine, for linked data based on the Rete match. *AImWD*, 2012.

[39] Neo Technology. Neo4j. `http://neo4j.org/`, 2013.

[40] Mikko Rinne. SPARQL update for complex event processing. In *ISWC'12*, volume 7650 of *LNCS*. 2012.

[41] Marko A. Rodriguez and Peter Neubauer. Constructions from dots and lines. *CoRR*, abs/1006.2361, 2010.

[42] Markus Scheidgen. How Big are Models – An Estimation. Technical report, Department of Computer Science, Humboldt Universität zu Berlin, 2012.

[43] Sherif Sakr. Supply cloud-level data scalability with NoSQL databases. `http://www.ibm.com/developerworks/cloud/library/cl-nosqldatabase/index.html`, March 2013.

[44] Andrés Taylor and Alistair Jones. Cypher Query Language. http://www.slideshare.net/graphdevroom/cypher-query-language, 2012.

[45] The Eclipse Project. Eclipse Modeling Framework website. http://www.eclipse.org/emf, October 2012.

[46] W3C. SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.

Appendix A

Graph formats

In this chapter, we provide examples for the different graph formats, including property graphs and RDF graphs. The examples describe a small instance model based on the TrainBenchmark's metamodel, shown on Figure A.1.

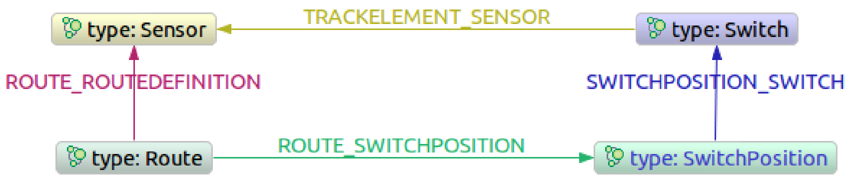

Figure A.1: *An example graph based on the TrainBenchmark's metamodel*

A.1 Property graph formats

A.1.1 GraphML

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
    http://graphml.graphdrawing.org/xmlns/1.1/graphml.xsd">
  <key id="type" for="node" attr.name="type" attr.type="string" />
  <graph id="G" edgedefault="directed">
    <node id="1">
      <data key="type">Sensor</data>
    </node>
    <node id="2">
      <data key="type">Route</data>
    </node>
    <node id="3">
      <data key="type">SwitchPosition</data>
    </node>
    <node id="4">
      <data key="type">Switch</data>
    </node>
    <edge id="0" source="2" target="1" label="ROUTE_ROUTEDEFINITION" />
    <edge id="1" source="2" target="3" label="ROUTE_SWITCHPOSITION" />
```

```
    <edge id="2" source="3" target="4" label="SWITCHPOSITION_SWITCH" />
    <edge id="3" source="4" target="1" label="TRACKELEMENT_SENSOR" />
  </graph>
</graphml>
```

Listing A.1: *A graph based on the TrainBenchmark's metamodel stored in GraphML format*

A.1.2 Blueprints GraphSON

```
{
  "vertices":[
    {
      "type":"Sensor",
      "_id":1,
      "_type":"vertex"
    },
    {
      "type":"Route",
      "_id":2,
      "_type":"vertex"
    },
    {
      "type":"SwitchPosition",
      "_id":3,
      "_type":"vertex"
    },
    {
      "type":"Switch",
      "_id":4,
      "_type":"vertex"
    }
  ],
  "edges":[
    {
      "_id":0,
      "_type":"edge",
      "_outV":2,
      "_inV":1,
      "_label":"ROUTE_ROUTEDEFINITION"
    },
    {
      "_id":1,
      "_type":"edge",
      "_outV":2,
      "_inV":3,
      "_label":"ROUTE_SWITCHPOSITION"
    },
    {
      "_id":2,
      "_type":"edge",
      "_outV":3,
      "_inV":4,
      "_label":"SWITCHPOSITION_SWITCH"
    },
    {
      "_id":3,
      "_type":"edge",
      "_outV":4,
```

```
      "_inV":1,
      "_label":"TRACKELEMENT_SENSOR"
    }
  ]
}
```


Listing A.2: *A graph based on the TrainBenchmark's metamodel stored in Blueprints GraphSON format*

A.1.3 Faunus GraphSON


```
{"type":"Sensor","_id":1,"_outE":[],"_inE":[{"_id":0,"_outV":2,"_label":"ROUTE_ROUTEDEFINITION"},{"_id":3,"_outV":4,"_label":"TRACKELEMENT_SENSOR"}]}
{"type":"Route","_id":2,"_outE":[{"_id":0,"_inV":1,"_label":"ROUTE_ROUTEDEFINITION"},{"_id":1,"_inV":3,"_label":"ROUTE_SWITCHPOSITION"}],"_inE":[]}
{"type":"SwitchPosition","_id":3,"_outE":[{"_id":2,"_inV":4,"_label":"SWITCHPOSITION_SWITCH"}],"_inE":[{"_id":1,"_outV":2,"_label":"ROUTE_SWITCHPOSITION"}]}
{"type":"Switch","_id":4,"_outE":[{"_id":3,"_inV":1,"_label":"TRACKELEMENT_SENSOR"}],"_inE":[{"_id":2,"_outV":3,"_label":"SWITCHPOSITION_SWITCH"}]}
```


Listing A.3: *A graph based on the TrainBenchmark's metamodel stored in Faunus GraphSON format*

A.2 Semantic graph formats

A.2.1 RDF


```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:swrl="http://www.w3.org/2003/11/swrl#"
  xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

<rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Ontology"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#Segment">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#Trackelement"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#Switch">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#Trackelement"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#1">
```

```
  <rdf:type rdf:resource="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#Sensor"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#2">
  <rdf:type rdf:resource="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#Route"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#3">
  <rdf:type rdf:resource="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#Switch"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#4">
  <rdf:type rdf:resource="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#SwitchPosition"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#3">
  <TrackElement_sensor rdf:resource="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#1"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#4">
  <SwitchPosition_switch rdf:resource="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#3"/>
</rdf:Description>

<rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#2">
  <Route_routeDefinition rdf:resource="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#1"/>
  <Route_switchPosition rdf:resource="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#4"/>
</rdf:Description>

</rdf:RDF>
```

Listing A.4: *A graph based on the TrainBenchmark's metamodel stored in RDF format.*