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 25, 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 ~~generációs,~~ 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

Chapter 1

Introduction

This chapter introduces the report's context. It defines the main problems and summarizes the goals of our research.

1.1 Context

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.

Models representing sensor data, reverse engineered software models (e.g. abstract syntax trees of existing source code) and geospatial models can contain well over 10^9 modeling elements [65]. The dramatic increase in complexity 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.

1.2 Problem Statement and Requirements

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 [34]) 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 [43]). 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 [68]). 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 [59, 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 [44]) has crucial conceptual and technological differences to traditional metamodeling languages such as Ecore [68]. Additionally, while there are initial efforts to overcome the mapping issues between the MDE and Linked Data worlds [47], even the most sophisticated NoSQL storage technologies lack efficient and mature support for executing expressive queries *incrementally*.

1.3 Objectives and Contributions

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 [41] that can scale up from a single-workstation tool to a cluster to handle very large models and complex queries efficiently.

The main contributions of this report is a *novel architecture* for a distributed, incremental query engine. We developed a *working prototype* and designed a *benchmark environment* to evaluate the scalability characteristics of the system.

We conducted benchmarks with different storage backends and query engines. The analysis of the results confirmed the feasibility of the approach.

1.4 Structure of the Report

The report is structured as follows. Chapter 2 introduces the background technologies and the motivation for building a distributed, incremental graph pattern matcher. Chapter 3 provides an overview of current a single-node incremental pattern macher, EMF-INCQUERY. Chapter 4 shows an initial performance evaluation in the context of on-the-fly well-formedness validation of software design models. Chapter 5 discusses the related work. Chapter 6 concludes the report and presents our future plans.

Chapter 2

Background Technologies

Developing 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 looked for technologies that can form the building blocks of a distributed, scalable model repository and pattern matcher. These technologies must been designed with scalability in mind and deployed in large-scale distributed systems successfully.

Usually, instance models are graph-like data structures. Therefore, we looked for scalable graph databases. In this context, scalability requires distributed storage and querying capabilities.

During the early phase of the research, we studied the architecture and limitations of the candidate systems. For databases, we inspected the data sharding strategies, consistency guarantees and transaction capabilities, along with the API and query methods. We also checked the systems' support for asynchronous processing, notification and messaging mechanisms.

In this chapter, we introduce the concepts and technologies that can form the basis of a scalable, distributed, asynchronous system.

2.1 Big Data and the NoSQL Movement

Since the 1980s, database management systems based on the relational data model [35] dominated the database market. Relational databases have a number of important advantages: precise mathematical background, understandibility, mature tooling and so on. However, due to their rich feature set and the strongly connected nature of their data model, relational databases often have scalability issues [53, 67]. They are typically optimized for transaction processing, instead of data analsysis (see *data warehouses* for an exception). In practice, these render them impractical for a number of use cases, e.g. running complex queries on large data sets.

In the last decade, large organizations struggled to store and process the huge amounts of data they produced. This problem introduces a diverse palette of scientific and engineering challenges, 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] [19]. ~~During the development of INCQUERY-D's prototype~~, we experimented with numerous NoSQL databases.

2.2 Concepts

This section introduces the most important concepts used in this report.

2.2.1 Metamodeling

Metamodeling is a methodology for the definition of modeling languages. A metamodel specifies the abstract syntax (structure) of a modeling language. Metamodels are expressed using a metamodeling language that itself is a modeling language. The metamodel can also be interpreted as the object-oriented data model of the language under design.

2.2.2 Graph Data Models

Along the well-known and widely used relational data model, there are many other data models. NoSQL databases are often categorized based on their data model (e.g. key–value stores, document stores, column families). In this report, we focus on *graph data models.*

The graph is a well-known mathematical concept widely used in computer science. For our work, it is important to distinguish between different graph data 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*). To allow different connections, we may label the edges (*labeled graph*).

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

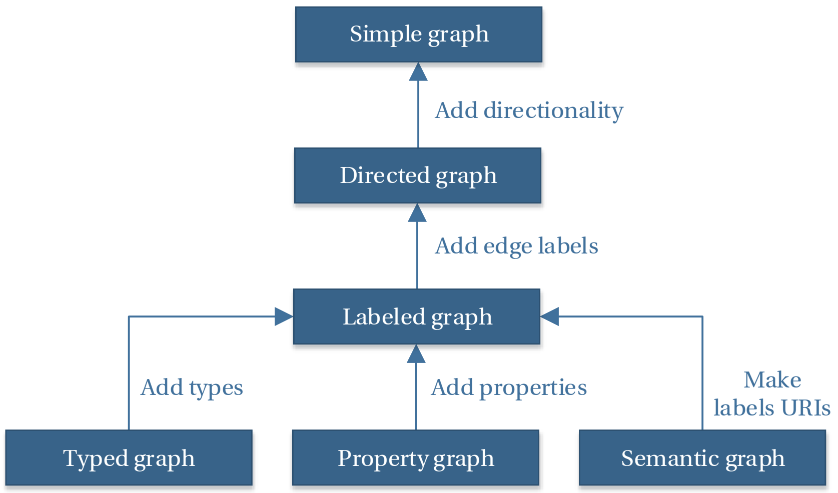

Figure 2.1: *Different graph data models (based on [64])*

*Typed graph*s introduces types for vertices. *Property graph*s (sometimes called *attributed graphs*) 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`. *Semantic graph*s use URIs (Uniform Resource Identifiers) instead of labels, otherwise they have similar expressive power as labeled graphs.

These graph models are found in many languages and environments. In the following, we will present the ones most important for this report: the Ecore metamodeling language, the TinkerPop framework and the Resource Description Framework (RDF).

Ecore

Ecore is the metamodeling language used by EMF. It has been developed in order to provide an approach for metamodel definition that supports the direct implementation of models using a programming language. The main rationale in introducing Ecore separately that it is the *de facto* standard metamodeling environment of the industry, and several domain-specific languages are defined using this formalism.

Ecore models can express *typed graphs* and *property graphs* as well.

Figure 2.2 illustrates the core elements of the Ecore approach. The full metamodel can be found in the EMF documentation [39]. The most important elements are the following.

- `EClass` models classes (or concepts). `EClasses` are identified by name and can have several attributes and references. To support inheritance, a class can refer to a number of *supertype* classes.

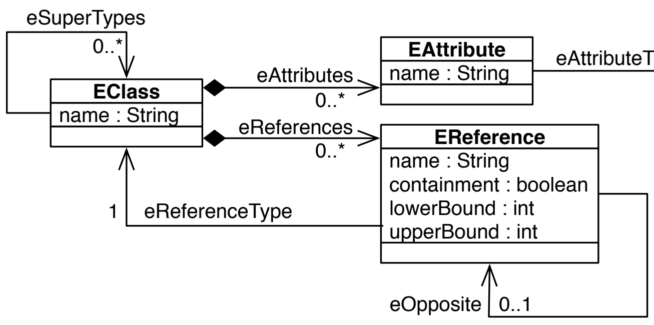

Figure 2.2: *The Ecore kernel, a simplified subset of the Ecore metamodel*

- `EAttribute` models attributes, that contain data elements of a class. They are identified by name and have a *data type*.
- `EDataType` is used to represent simple data types that are treated as atomic (their internal structure is not modeled). Data types are also identified by their name.
- `EReference` represents a unidirectional association between `EClasses` and as identified by a name. Lower and upper multiplicities can be specified. It is also possible to mark a reference as a *containment* that represents composition relation between elements. If a bidirectional association is needed, it should be modeled as two `EReference` instances that are mutually connected via their *opposite* references.

The rest of the Ecore metamodeling language contains utility elements, common supertypes that support the organization and hierarchization of the models ~~but the main metamodeling part has been covered here.~~

TinkerPop framework

The *TinkerPop* framework is an open-source software stack for graph storage and processing [28]. 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 (Section 2.3.2), Titan (Section 2.3.3), DEX [24], InfiniteGraph [20] and OrientDB [21].

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. For example, if John's father is Jack and Jack's father is Scott, we may run the traversals shown on Listing 2.1.

```
gremlin> g.V('name', 'John').out('father')
```

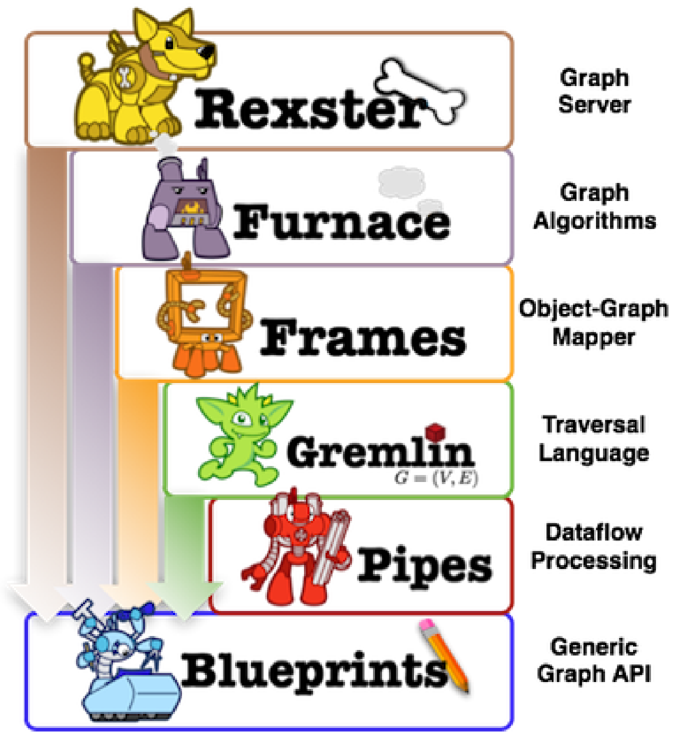

Figure 2.3: *The TinkerPop software stack [11]*

```
==>Jack
gremlin> g.V('name', 'John').out('father').out('father')
==>Scott
```

Listing 2.1: *Simple Gremlin queries*

Gremlin is based on *Pipes*, TinkerPop's dataflow processing framework. Besides traversing, Gremlin is 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 2.3.

Resource Description Framework

The Resource Description Framework (RDF) is a family of W3C (World Wide Web Consortium) specifications originally designed as a *metadata data model*.

The RDF data model is based on the idea of making statements about *resources* in the form of triples. 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 typically stored in *triplestores*, specialized databases tailored to store and process triples efficiently. Also, some triplestores are capable of *reasoning*, i.e. inferring logical consequences from a set of facts or axioms. Triplestores are mostly used in semantic technology projects.

Triplestores are usually queried via the RDF format's query language, SPARQL (recursive acronym for SPARQL Protocol and RDF Query Language).

The RDF data model is capable of expressing *semantic graph*s. Although the semantic graph data model has less expressive power than the property graph data model, by introducing additional resources for each property, a property graph can be easily mapped to RDF.

Mapping Ecore to Other Data Models

Our intention to reuse EMF-INCQUERY for building INCQUERY-D required us to map EMF's metamodel, Ecore to the domain of property graphs and RDF models.

Ecore concept	**Property graph concept**	**RDF concept**
`EClass` instance	nodes' `type` property	`rdfs:Resource`
`EAttribute` instance	nodes' property names	`rdf:Property`
`EReference` instance	edge label	`rdf:Property`
`EDataType` instance	Java primitive types	`rdfs:Datatype`

Table 2.1: *Mapping Ecore to property graphs and RDF*

2.2.3 Sharding

To provide scalable persistence and processing for large amounts of data, the data has to be split between multiple computers. This process is known as *data sharding*. *Graph sharding* is a particularly difficult problem due to the strongly connected and mutable nature of graphs. Efficient sharding of graphs is still an ongoing research problem [54].

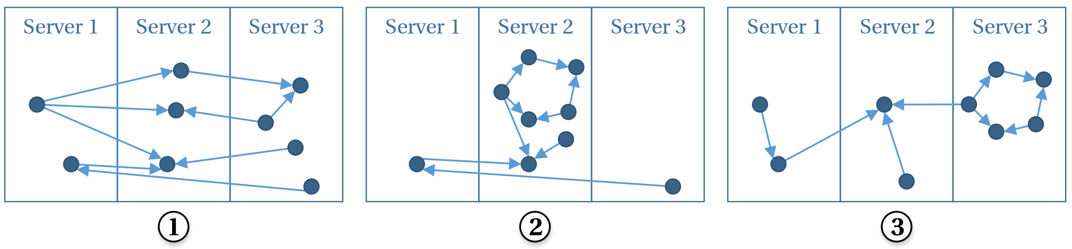

Figure 2.4: *Different partitionings of the same graph*

To illustrate the problem, Figure 2.4 shows different partitionings of the same graph in a three-node cluster. In case ① most edges run between servers and are therefore expensive to traverse. In case ②, *Server 2* is overloaded, taking more than three quarters of the total load. Case ③ presents a more balanced sharding of the graph. Unfortunately, for large graphs, balanced sharding is hard to achieve in practice.

Most graph partition problems NP-hard and practical solutions to these problems could be derived using heuristics and approximation algorithms [40]. Unfortunately, open-source database implementations lack support for such algorithms.

2.2.4 Query Languages and Evaluation Strategies

In the context of this report, a query defines a *graph pattern*. The result of the query is a *set of subgraphs* of the original graph. Graph patterns are useful for identifying patterns in a set of connected data elements. They are especially widely used in the context of model-driven engineering for defining well-formedness validation constraints and graph transformations.

Query Language

Queries can be defined in both imperative and declarative languages. The theoretical basis for most declarative query languages is first order logic. Both tuple relational calculus and relational algebra (widely used in query processing) are offshoots of first-order logic. Relational calculus defines the , relational algebra .

Query Evaluation Strategies

Query engines can be divided into two core categories: *search-based* and *incremental* engines. The main difference between these approaches is the way they re-evaluate queries. While search-based engines process the whole data set (i.e. not just the data elements affected by the change), incremental engines utilize some data structures to be able to re-evaluate the query based on the change set.

2.2.5 Miscellaneous

Asynchronous Parallel Processing with MapReduce

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.

MapReduce is often used for sorting, filtering and aggregating data sets. It is also used for fault-tolerant, distributed task execution.

The Column Family Data Model

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 problems associated with NULL values.

2.3 Graph Storage Technologies

In this section, we compare different graph storage technologies by discussing their architecture and data model. We inspect their query languages, with particular emphasis on the support of distributed operations. We also present the systems' sharding strategies for distributed storage.

2.3.1 EMF Technologies

Eclipse is a free, open-source software development environment and a platform for plug-in development. Eclipse comes with its own modeling technologies called EMF (Eclipse Modeling Framework). EMF's primary goals are application design and code generation.

Architecture

EMF models can be persisted as XMI (XML Metadata Interchange) documents. By design, EMF models cannot be fragmented, i.e. they can only be used if they fit to a computer's main memory. There are different model repositories and persistence frameworks which can handle large EMF models [66].

- CDO (Connected Data Objects), a distributed shared model framework for EMF models and metamodels [38]. CDO provides an object-relational mapping from Ecore to databases.

- Morsa [60] is a distributed model repository based on MongoDB [17], a popular NoSQL database management system.

Data Model

EMF uses the Ecore data model, discussed in Section 2.2.2.

Sharding

Due to the nature of XML documents, EMF models serialized to a single XMI document cannot be sharded. CDO does not support automatic sharding, however Morsa does so by using MongoDB's sharding mechanism.

Query Language and Evaluation

OCL OCL (Object Constraint Language) is a declarative query language to describe well-formedness constraints on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model.

EMF-INCQUERY EMF-INCQUERY [33] is an Eclipse project developed by the Fault Tolerant Systems Research Group in the Budapest University of Technology and Economics. It provides IQPL (INCQUERY Pattern Language), a declarative language to express queries over EMF models in the form of graph patterns. With the language the user can express combined queries, negative patterns, checking property conditions, simple calculations, calculate disjunctions and transitive closures, etc. on top of the models. The goal of EMF-INCQUERY is to provide *incremental query evaluation*.

EMF-INCQUERY's performance was subject to numerous benchmarks, with some being presented in Section 4.2.

2.3.2 Neo4j

Neo4j, developed by Neo Technology, is the most popular NoSQL graph database. Neo4j is one of the most mature NoSQL databases. It is well documented and provides ample tooling, including an Eclipse-based visualization application, Neoclipse [18].

Architecture

Neo4j can be deployed in two scenarios. In *embedded mode*, it runs in the same JVM (Java Virtual Machine), as the client application. In this setup, the database cannot be accessed by other applications. In *server mode*, the database can serve requests from multiple clients over a REST (Representational State Transfer) interface.

Data Model

Neo4j implements the TinkerPop framework's Blueprints property graph data model. Neo4j is capable of loading graphs from GraphML [25] and Blueprints GraphSON [14] formats (see Section A.1 for examples).

Sharding

Instead of sharding, Neo4j only supports replication of data to create a highly available cluster. This implies serious scalability limitations to the system. Neo4j's developers make serious efforts to improve the scalability of the database in an ongoing project called Rassilon [3].

Query Language and Evaluation

Neo4j can be queried in various ways. When deployed in embedded mode, the application can use its Java-based core API. In both embedded and server mode, Neo4j provides two query languages. The first is the TinkerPop framework's imperative Gremlin language, primarily targeted for graph traversals. The second is Neo4j's own declarative query language for graph pattern matching, Cypher.

2.3.3 Titan

Titan is a distributed, scalable graph database from Aurelius, the creators of the TinkerPop framework.

Architecture

Titan is not a standalone database, instead, it builds on top of existing NoSQL database technologies and leverages Hadoop's MapReduce capabilities. Titan supports various storage backends, including Cassandra and HBase. In the following, we shortly cover the technologies Titan builds upon. Both Titan and it's dependencies are open-source software, written in Java.

Hadoop Hadoop is a distributed data processing framework inspired by Google's publications about MapReduce [36] and the Google File System [42]. 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). A typical 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 2.5).

HDFS The Hadoop Distributed File System (HDFS) is an distributed file system, inspired by the Google File System and written specifically for Hadoop [7]. Unlike other distributed file systems (e.g. Lustre [16]), which require expensive hardware components, HDFS was designed to run on commodity hardware. HDFS tightly integrates with Hadoop's architecture (Figure 2.5).The *NameNode* is responsible for storing the metadata of the files and the location of the replicas. The data is stored by the

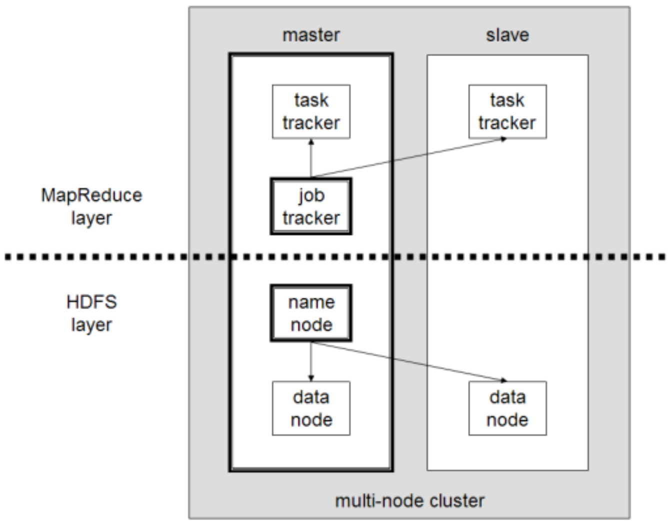

Figure 2.5: *Hadoop's architecture [56]*

*DataNode*s.

HBase HBase [8] is an 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.

Cassandra Cassandra is one of the most widely used NoSQL databases [6]. Originally developed by Facebook [55], Cassandra is now an Apache project. Cassandra's a column family database with advanced fault-tolerance mechanisms. It allows the application to balance between availability and consistency by allowing it to tune the consistency constraints. Cassandra is used mainly by Web 2.0 companies, including Digg, Netflix, Reddit, SoundCloud and Twitter. It is also used for research purposes at CERN and NASA [22].

Data Model

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.

Sharding

Titan uses the storage backend's partitioner, e.g. Cassandra's hash-based 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 as of October 2013, but not yet avail-

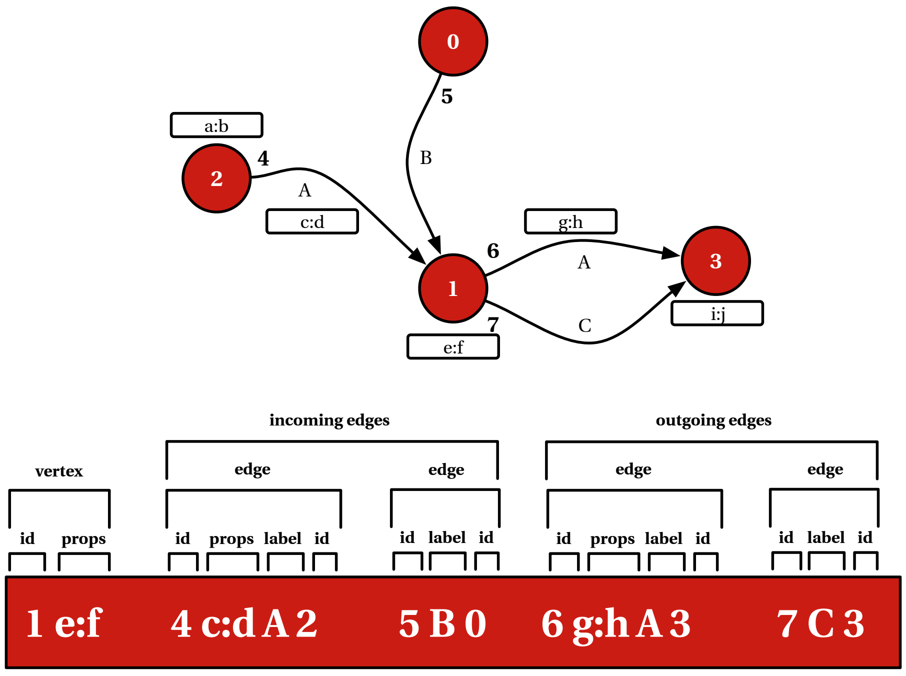

Figure 2.6: *Graph vertex mapped by Titan to a row in a Cassandra database*

able.

Query Language and Evaluation

Titan supports the TinkerPop framework's Gremlin query language. Gremlin/Pipes utilizes a depth-first search algorithm.

Faunus Although Titan was designed with scalability in mind, its query engine does not work in a parallel way. Also, it is unable to cope with queries resulting in millions of graph 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 Section A.1.3 for an example. Unlike the Gremlin implementation in Neo4j and Titan, the implementation in Faunus is based on breadth-first search. It is important to note that Faunus always traverses the whole graph and does not use its indices. This makes retrieving nodes or edges by type very slow (see our typical workload in Section 3.3.2).

2.3.4 4store

4store is an open-source, distributed triplestore created by Garlik [4]. Unlike the other tools discussed earlier, 4store is written in C. 4store is primarily applied for semantic web projects.

Architecture

4store was designed to work in a cluster with high-speed networks. 4store server instances are capable of discovering each other using the Avahi configuration protocol [10]. 4store offers a command-line and a HTTP server interface.

Data Model

4store's data model is an RDF graph. It supports RDF/XML input format, which is processed using the Raptor RDF Syntax Library [61].

Sharding

Similar to Titan's partitioning, 4store's *segmenting* mechanism distributes the RDF resources evenly across the cluster. 4store also supports replication by *mirroring* tuples across the cluster.

Query Language and Evaluation

4store supports SPARQL queries with the Rasqal RDF Query Library [62].

2.3.5 Overview and Evaluation of Graph Storage Technologies

Technology	**Data model**	**Distributed operation**	**Sharding**	**Queries**	**Identifier generation**
EMF	Ecore	Differs	Differs	OCL, IQPL	**Automatic**
4store	RDF	Manual	**Automatic**	SPARQL	Manual
Neo4j	Property graph	Manual	Manual	Cypher	Manual
Titan	Property graph	**Automatic**	**Automatic**	Gremlin	**Automatic**

Table 2.2: *Overview of database technologies*

Table 2.2 summarizes the relevant characteristics of the aforementioned database management systems. Accoding to these, Titan provides the most complete feature set. 4store and Neo4j lack important features like automatic identifier generation, which has to be implemented in the client application. Neo4j also misses automatic sharding, which seriously hinders its scalability potential. EMF's distributed operation and sharding capabilities depend on the actual model repository and database backend being used.

2.4 Asynchronous Messaging: Akka

Most distributed, concurrent systems use a messaging framework or message queue service. The INCQUERY-D system also requires a distributed, asynchronous messaging framework. For this purpose, we used the Akka framework.

Akka is an open-source, fault-tolerant, distributed, asynchronous messaging framework developed by Typesafe [5]. 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.

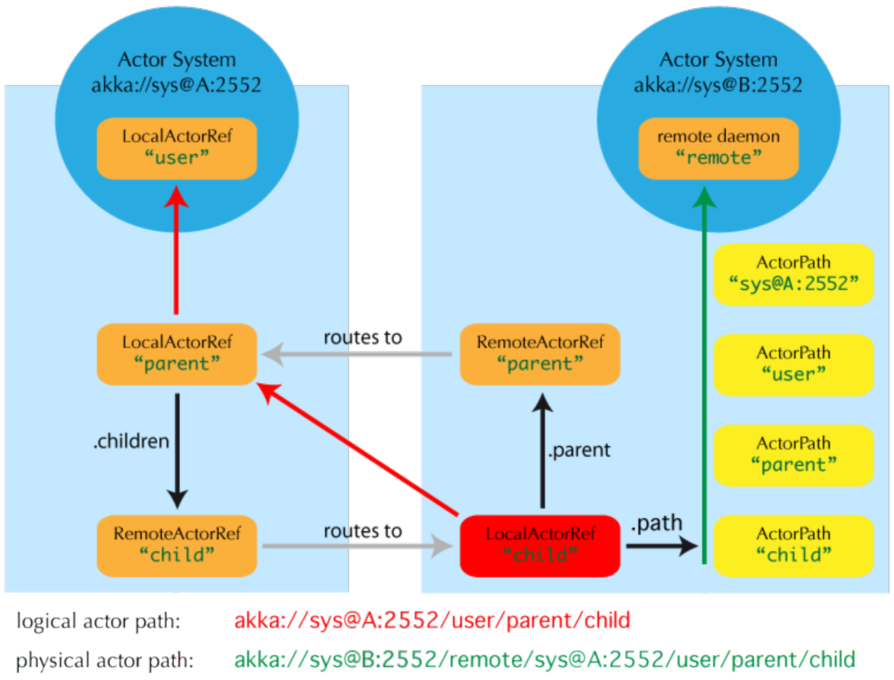

Figure 2.7: *Deploying a remote actor in Akka [5]*

Akka is based on the actor model [46] and provides built-in support for remoting. Unlike traditional remoting solutions, e.g. Java RMI (Remote Method Invocation) and CORBA (Common Object Request Broker Architecture), the remote and local interface is the same for each actor. Actors have both a logical and a physical path (Figure 2.7). This way, they can be transparently moved between machines on the network.

As of October 2013, the latest version (Akka 2.2) also supports *pluggable transport* to use various transports to communicate with remote systems [5]. For serializing the messages, Akka supports different frameworks, including Java's built-in serialization, Google Protobuf [23] and Apache Thrift [9].

2.5 Motivation for Horizontal Scaling

EMF-INCQUERY is proven to be effecient for incremental query evaluation on small to medium-sized models (in the order of magnitude of 10^6). However, model-driven engineering challenges often present large models, with 10^9 or more elements [65].

Due to the Rete algorithm's memory consumption (Section 3.1.2), EMF-INCQUERY cannot handle large models efficiently [34]. A trivial solution would be to use *vertical scaling*, i.e. putting more memory in the workstation. Unfortunately, this approach is not feasible due to nature of Java's memory management. The Garbage Collector (GC) cannot handle heap sizes larger than 10 GB efficiently, thus introducing long pauses in the application [26].

This problem is well-known in the Java community. There are alternative Java Virtual Machines (JVMs) with specialized Garbage Collectors, like Azul Systems' JVM. However, the Azul JVM is a proprietary product and has specific hardware requirements. Also, this does not solve the scaling problem entirely – the model size is still limited by the total amount memory in a single computer.

Instead of vertical scaling, we decided to opt for *horizontal scaling*. As described in Section 2.1, distributed non-relational databases have been gaining momentum in the last years. For persisting models, we inspected graph databases, but found that their query layer does not scale well in a distributed environment (Section 4.6). This is a serious problem for MDE, where the queries are typically more complex than in traditional, transactional database management.

Therefore, we decided to design a stand-alone distributed, scalable query engine based on NoSQL and semantic web databases.

Chapter 3

Overview of the Approach

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 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 Incremental Query Evaluation

Some queries, e.g. well-formedness constraints in MDE are evaluated many times, while the data set they are evaluated on only changes to a small degree. In these cases, the idea of incremental query evaluation arises naturally: to speed up queries, we should not start the evaluation all over again. Instead, we should rely on the (partial) results derived during the previous executions of the query and process only the changes that occured.

In practice, incremental query evaluation algorithms typically use data structures for caching the interim results. This means that they usually consume more memory, in other words, they trade memory consumption for execution speed. This approach, called *space–time tradeoff*, is well-known and widely used in computer science.

In the following, we provide an overview of the *Rete algorithm*, which forms the theoretical basis of EMF-INCQUERY and INCQUERY-D.

3.1.1 Incremental Pattern Matching Algorithms

Numerous algorithms were invented for the purpose of incremental pattern matching. Mostly, these algorithms originate from the field of rule-based expert systems.

One of the most well-known is the *Rete algorithm*, which creates a propagation network, which stores the partial matches found in the graph[1]. TREAT [57] aims at minimizing memory usage by using only indexers and dropping partial results, while having the same algorithmic complexity as Rete. Another candidate is the LEAPS [30] algorithm, which is claimed to provide better space–time complexity. However, we found that LEAPS is difficult to understand and implement even on a single workstation, not to mention the distributed case.

Rete has many improved versions (e.g. Rete II, Rete III, Rete-NT), however, unlike the original algorithm, these are not publicly available. Because the original Rete algorithm is well-understood by the EMF-INCQUERY team, we decided to build INCQUERY-D on the same foundation. Experimenting with improved versions or alternative approaches is subject to future work.

3.1.2 The Rete Algorithm

The algorithm was originally created by Charles Forgy [41] for rule-based expert systems. Gábor Bergmann adapted the algorithm for EMF models and added many tweaks and improvements to it [31].

The Rete algorithm defines an asynchronous network of communicating nodes (Figure 3.1). This is essentially a dataflow network, with two types of nodes. Change notification objects (*tokens*) are propagated to intermediate *worker nodes* that perform operations known from relational algebra, like projection (π), selection (σ), join ($\bowtie$) and antijoin ($\triangleright$) operations. The worker nodes store partial query results in their own memory. In contrast, *production nodes* are terminators that provide an interface for fetching query results and also their change sets (*deltas*).

The Rete network is built on top of type-specific indexers, which are reponsible for providing quick lookups and generating notifications for the worker nodes.

[1] *Rete* is Latin for *net*.

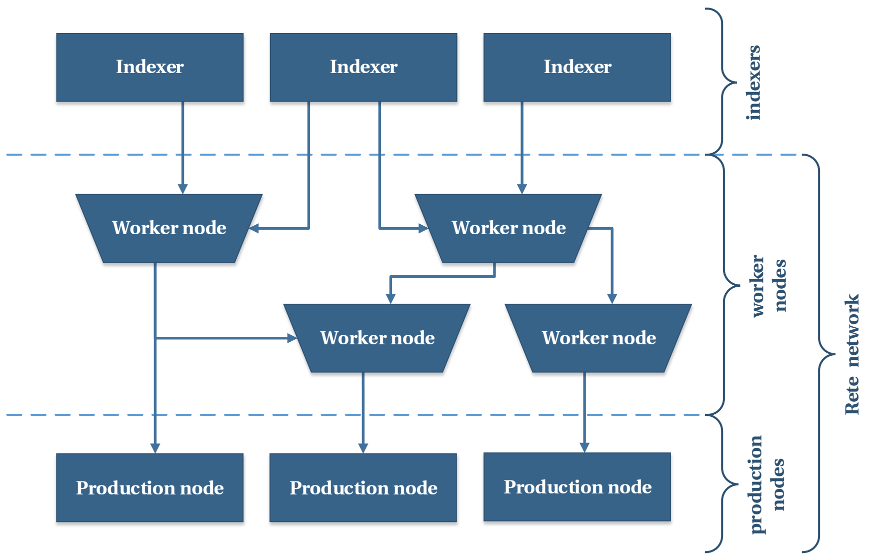

Figure 3.1: *The structure of the Rete propagation network*

3.2 Incremental Pattern Matching on a Single Node: EMF-IncQuery

In the following, we will overview the architecture of a *single-node* incremental pattern matcher, specifically EMF-IncQuery.

3.2.1 Architecture

The Rete algorithm forms the foundation of EMF-IncQuery's query engine. Figure 3.2 shows the architecture of EMF-IncQuery and the Rete network's role in the system.

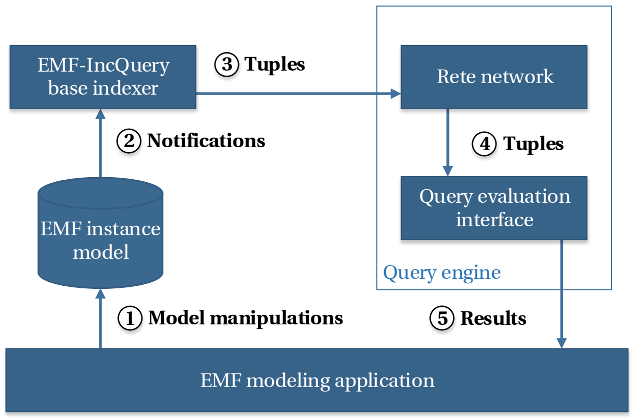

Figure 3.2: EMF-IncQuery*'s architecture*

A typical model transformation sequence is the following. The modeling application manipulates the EMF instance model ①. The model sends notifications to EMF-IncQuery's base indexer ②. The indexer propagates the modified tuples to the

Rete network as update messages ③, which processes the updates and sends the resulting tuples to the query evaluation interface ④. The modeling application can retrieve the results from the interface ⑤.

3.2.2 Indexing and Initialization

Indexing is a common technique for decreasing the execution time of database queries. In MDE, *model indexing* has a key role to high performance model queries. As MDE primarily uses a metamodeling infrastructure, all queries utilize some type attribute. Typical elementary queries are listed below.

- Retrieving all node instances of a given type (e.g. get all nodes with the type `Person`).
- Retrieving all edge instances of a given label (e.g. get all edges with the label `child`).
- Retrieving a given node's all incoming and/or outgoing edges of a given type (e.g. get all outgoing `child` edges of a given node).
- Reverse navigation: retrieving the node on the other end of an edge (e.g. the `child` relation is identical to the inverse of the `parent` relation).

EMF-IncQuery uses the EMF API to run this queries efficiently.

3.2.3 Data Representation and Storage

EMF-IncQuery works on in-memory EMF models. The Rete network represents the data in *tuples*. Basically, the network's tuples can contain two sorts of values: (i) pointers to an EMF model, (ii) Ecore scalar values (`EString`, `EInt`, etc. instances). This data representation principle intends to keep the Rete network's size as small as possible, while allowing efficient processing. Because of the tuple representation, various operations, e.g. projection (π) and join ($\bowtie$), can be simply defined using tuple masks [31].

3.2.4 Notification Mechanisms

Model change notifications are required by incremental query evaluation, thus model changes are captured and their effects are propagated in the form of *notification objects* (NOs). The notifications generate *tokens* that keep the Rete network's state consistent with the model.

3.2.5 Termination Protocol

As the Rete algorithm's change propagation is asynchronous, the system must also implement a *termination protocol* to ensure that the query results can be retrieved consistently with the model state after a given transaction (i.e. by signaling when the update propagation has been terminated).

3.2.6 Degrees of Freedom

For a given model, the system's performance for a query is mainly determined by the layout of the generated Rete network. Similarly to relational query optimization, we can also optimize the Rete network's layout. Currently, EMF-INCQUERY supports basic optimizations. It utilizes node sharing, i.e. it detects if two Rete nodes would store the same partial matches and merges them to a single node. More details are available in [32].

3.3 Extensions for Distributed Scalability: INCQUERY-D

Developing a distributed, scalable, incremental pattern matcher introduces numerous challenges. In the following, we will cover the INCQUERY-D's architecture and its main extensions to EMF-INCQUERY.

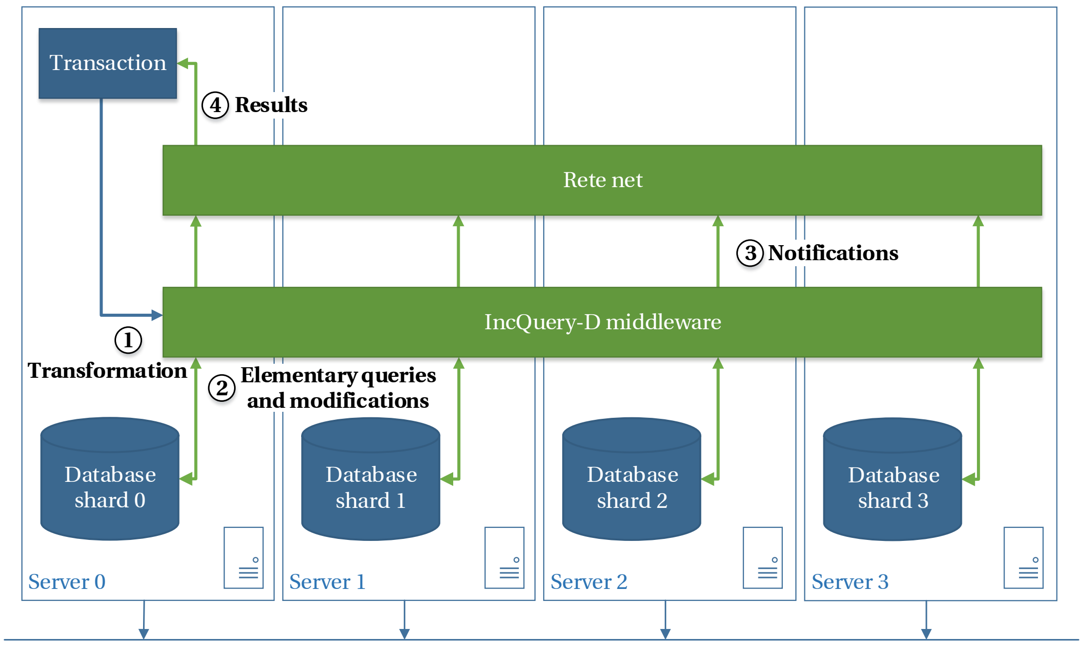

Figure 3.3: INCQUERY-D*'s architecture on a four-node cluster*

3.3.1 Architecture

The INCQUERY-D architecture in an example configuration is shown in Figure 3.3. 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 model (Section 3.3.3). The client application communicates with the *middleware* ①. The middleware provides a unified API for accessing the database ②. It also sends change notifications ③ (Section 3.2.4) to the production network and retrieves the query results from the production network ④ . The *production network* is implemented with a distributed Rete network which provides incremental query evaluation (Section 3.1.2).

3.3.2 Indexing and Initialization

To process the queries requires for indexing efficiently, 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 the bottom layer of the Rete production network. During initialization, these indexers are filled from the database backend (Figure 3.3 ②). In order to reduce the initialization time, the underlying storage layer must be able to process these queries efficiently.

3.3.3 Data Representation and Storage

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 instance models (Section 2.2.2) could be supported, as long as they can be mapped to an efficient and distributed storage backend.

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 can be executed efficiently.

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).

To support different data models, we only have to supply the appropriate connector class to INCQUERY-D's middleware. The current prototype supports 4store, Neo4j and Titan.

3.3.4 Notification Mechanisms

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 capable of generating notification events, but the events are only propagated in a single JVM (Java Virtual Machine). Implementing distributed notifications would require us to extend the `EventGraph` class and use a messaging framework. This is subject to future work (see Section 6.3).

Because the lack of support for distributed notifications, in INCQUERY-D's prototype, notifications are controlled by the middleware by providing a facade for all model manipulation operations (Figure 3.3 ③). The notification messages are propagated through the Rete network via the Akka messaging framework.

3.3.5 Termination Protocol

INCQUERY-D's ~~current~~ termination protocol works by adding a stack to the message. The stack registers each Rete node the message passes through. After the message reaches the production node, the termination protocol starts. Based on the content of the stack, acknowledgement messages are propagated back on the network. When all relevant indexer nodes (where the original notification token(s) started from) receive the acknowledge messages, the termination protocol finishes.

3.3.6 ~~Degrees of Freedom~~

The Rete algorithm (Section 3.1.2) utilizes both indexing and caching to provide fast incremental query evaluation. INCQUERY-D's horizontal scalability is supported by the distribution of the pattern matcher's Rete network. To enable this, the system must be able to allocate the Rete nodes to different hosts in a cloud computing infrastructure.

The deployment and configuration of a distributed pattern matcher involves many degrees of freedom, and design decisions. The overall performance of the system is influenced by a number of factors.

- For the storage layer, we may choose different database implementations due to the INCQUERY-D's backend-agnostic nature. In this report, we used property graph databases (Neo4j, Titan) and triplestores (4store).

- We may use different database sharding strategies (e.g. random partitioners or more sophisticated sharding methods based on domain-specific knowledge).

- Using query optimization methods, we can derive *Rete networks with different layouts* for the same query. The most efficient layout can be choosen based on both query and instance model characteristics, e.g. to keep the resource requirement of intermediate join operations to a minimum.

- We may choose different strategies to *allocate the Rete nodes* in the distributed system. The optimization strategy may choose to optimize local resource usage, or to minimize the amount of remote network communication. Note that in theory, this is *orthogonal* to the database's sharding strategy, i.e. these are two distinct level of distribution that do not directly depend upon each other. However, we expect that keeping the Rete network's type indexer nodes and the instances of the given type on the same server would improve the speed of the initialization and modification tasks significantly.

- We may implement *dynamic adaptability* to changing conditions. For example, 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.

3.4 Workflow

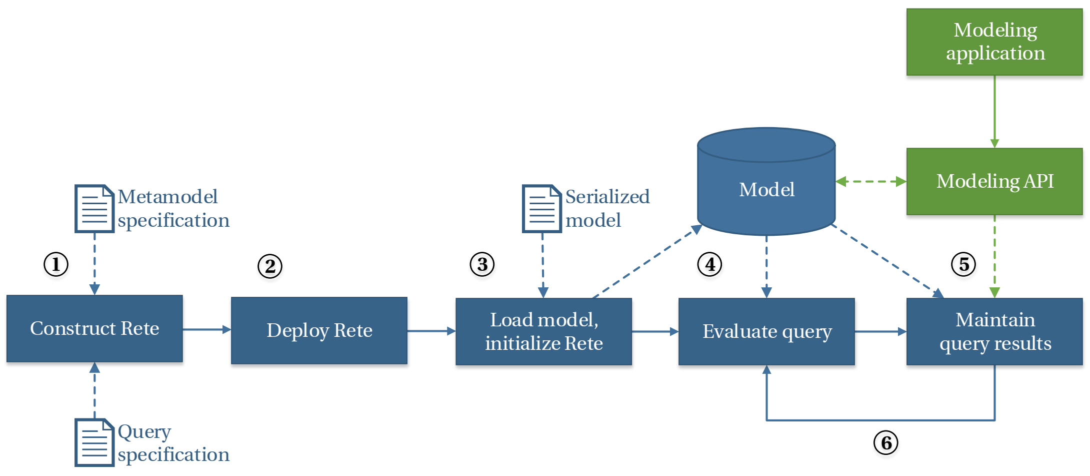

Figure 3.4: *The general workflow of incremental pattern matching with the Rete algorithm*

In the following, 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 3.4. First, we describe the workflow of EMF-INCQUERY and then emphasize the differences in INCQUERY-D's.

3.4.1 EMF-INCQUERY's Workflow

Based on the *metamodel* and the *query specification,* EMF-INCQUERY first constructs a Rete network ① and deploys it ②. It loads the model (from the persistent storage) to an *in-memory storage* ③ and traverses it to initialize the Rete network's indexers. The Rete network evaluates the query by processing the incoming tuples ④. If the modeling application modifies the model through the EMF API, the modifications are propagated the Rete network, hence keeping it in a consistent state ⑤. The query results can be retrieved from the Rete network ⑥. The modeling application may modify the model and reevaluate the query again.

3.4.2 INCQUERY-D's Workflow

By design, INCQUERY-D's workflow's steps are similar to EMF-INCQUERY's, discussed in Section 3.4.1. However, due to the system's distributed nature, they are more difficult to design and implement.

The main differences are the following. In INCQUERY-D, deploying the Rete network ② requires the deployment of remote actors (Section 2.4) on the servers. Both the Rete indexers and the database are distributed across the cluster. Hence, loading the model and initializing the Rete network needs network communication ③. The Rete network works using Akka's remote messaging feature. The query results can be retrieved from the Rete network (this may also require network communication) ④. The database shards can only be accessed through the middleware, which is reponsible for sending notifications to the Rete network's appropriate indexers. After the notifications are processed and the distributed termination algorithm finishes, the Rete network is in a consistent state ⑤. The results can be retieved by the client and it may modify the model an reevaluate the query again ⑥.

3.5 Tooling for INCQUERY-D

As mentioned earlier, we aimed to build INCQUERY-D on top of EMF-INCQUERY's pattern language (IQPL) and its Rete network generator. Because EMF-INCQUERY has an Eclipse-based user interface for defining and executing queries.

For INCQUERY-D, we plan to provide the same tooling environment. Also, for the allocation of Rete nodes, we created an Eclipse-based editor and viewer.

To aid the system's dynamic capabilities, we plan to develop a *runtime model-based dashboard* to monitor the state of INCQUERY-D's nodes. Currently, the INCQUERY-D tooling generates an architecture file (`arch`), which is used for deploying the distributed pattern matcher.

This file contains the Rete network's layout and its allocation in the cloud (as of now, the latter is defined manually). INCQUERY-D uses the architecture description for instantiating the Rete network and initializing the middleware (Figure 3.5).

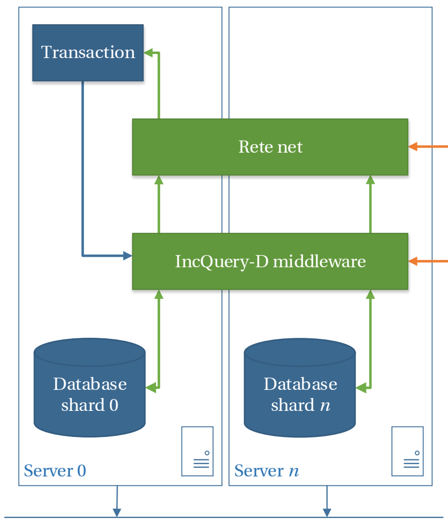

Figure 3.5: *Architecture of* INCQUERY-D *with a runtime dashboard*

To provide live feedback, we will adopt a *live* architecture model. The live model will provide real-time details about the systems' current state, including the local resources on each server, the Rete nodes' memory consumption and so on.

3.6 Elaboration of the Example

To demonstrate INCQUERY-D's approach, we elaborate an example in detail. We introduce a case study, then formulate a query and show the workflow that executes the distributed, incremental evaluation of the pattern defined by query.

3.6.1 Case Study: Railroad System Design

The example is built around an ~~imaginary~~ railroad system defined in the MOGENTES EU FP7 [69] project. The system's network is composed of typical railroad items, including signals, segments, switches and sensors. The complete EMF metamodel is shown on Figure 3.6. A subgraph of an instance model is shown on Figure 3.7.

We defined queries that resemble a typical MDE application's workload. In general, MDE queries are more complex than those used in traditional databases. They often

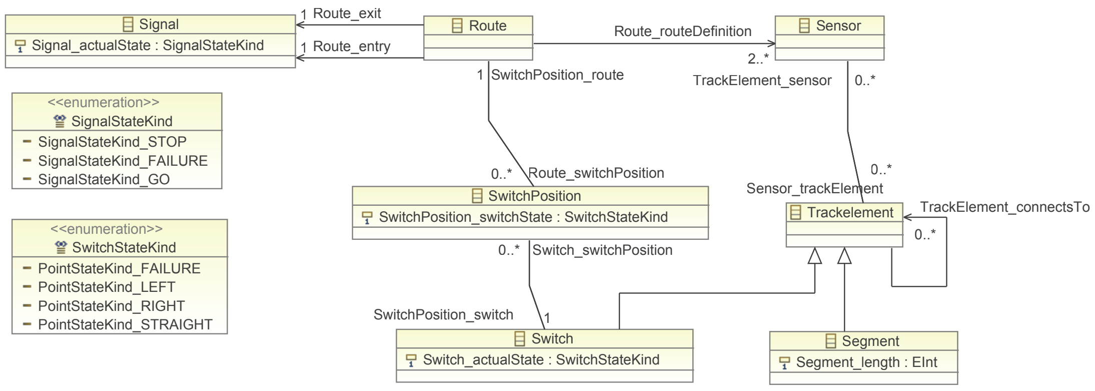

Figure 3.6: *The EMF metamodel of the railroad system*

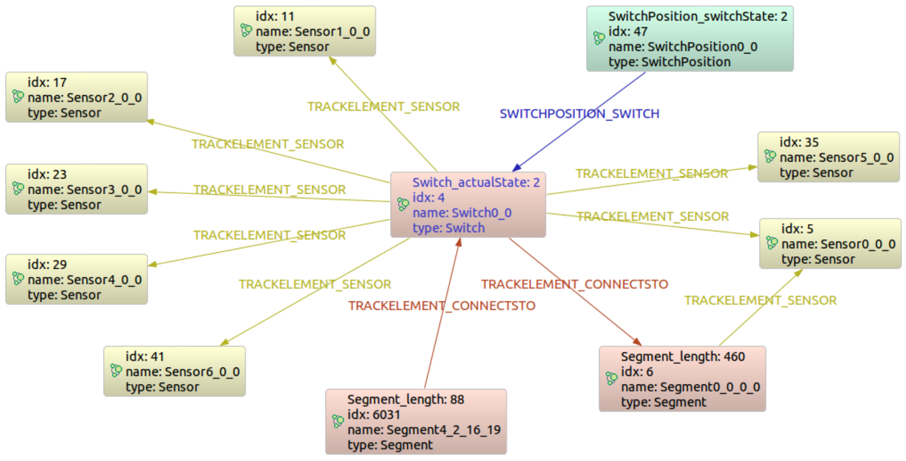

Figure 3.7: *A subgraph of a railroad system visualized*

define large patterns with multiple join operations. The queries look for violations of *well-formedness constraints* in the model. In this section, we discuss the *RouteSensor* query in detail.

RouteSensor

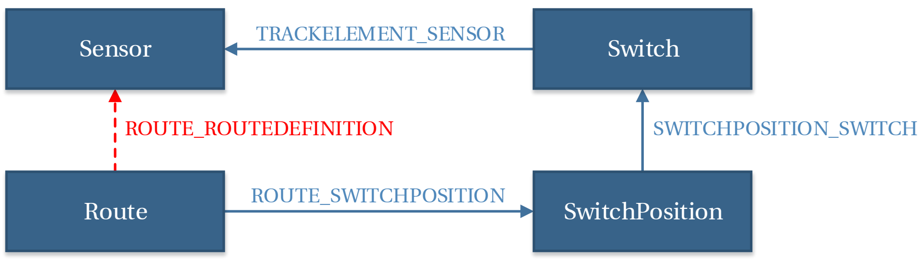

Figure 3.8: *Graphical representation of the RouteSensor query's pattern. The dashed red arrow defines a negative application condition.*

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 the RouteSensor query is shown on Figure 3.8. Basically, the RouteSensor query binds the type of the vertices, defines three edges and one negative edge, called NAC (negative application condition).

```
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*

The RouteSensor query in IQPL (INCQUERY Pattern Language) is shown on Listing 3.1. This query binds the variables (Sen, Sw, Sp, R) to the appropriate type. It defines the three edges as relationships between the variables and defines the negative application condition as a negative pattern (`neg find`).

For comparison, we also present the RouteSensor query in SPARQL (RDF's query language) on Listing 3.2. Here, the types are defined with the `rdf:type` predicate, while the edges are defined with base predicates. The negative application condition is defined with the `FILTER NOT EXISTS` construction[2].

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

[2]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.

```
    ?xSwitch base:TrackElement_sensor ?xSensor .

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

Listing 3.2: *The RouteSensor query in SPARQL*

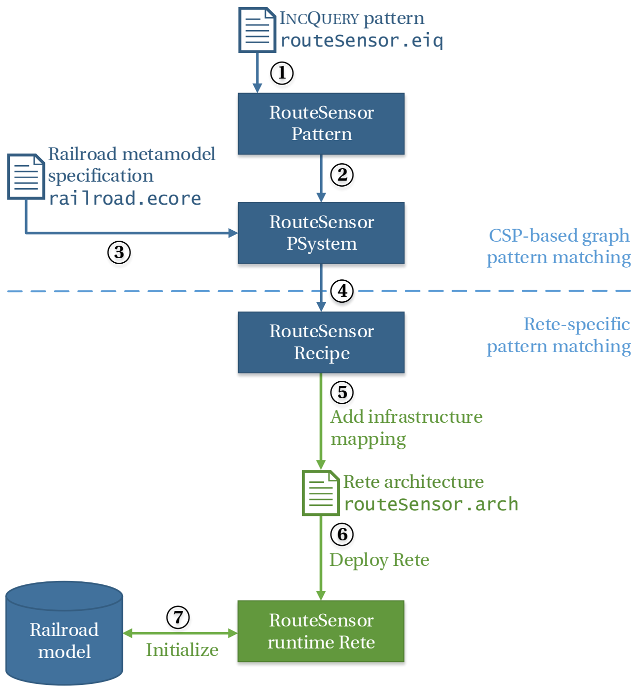

Figure 3.9: INCQUERY-D*'s workflow*

3.6.2 Workflow of the Example

Following the workflow defined in Section 3.4, we will cover the steps for deploying and operating a distributed pattern matcher. The actual workflow for the *RouteSensor* query is shown on Figure 3.9.

Constructing a Rete network

First, using EMF-INCQUERY's tooling, the query (`routeSensor.iqpl`, see Listing 3.1) is analyzed and parsed to an EMF model ①.

The metamodel (`railroad.ecore`) is shown on Figure 3.6. Based on the query ② and the metamodel ③ EMF-INCQUERY builds a *pattern system* (PSystem). The PSystem is translated to a Rete recipe, the system derives a Rete layout ④, that guarantees the satisfaction of the constraints. The Rete layout is shown on Figure 3.10.

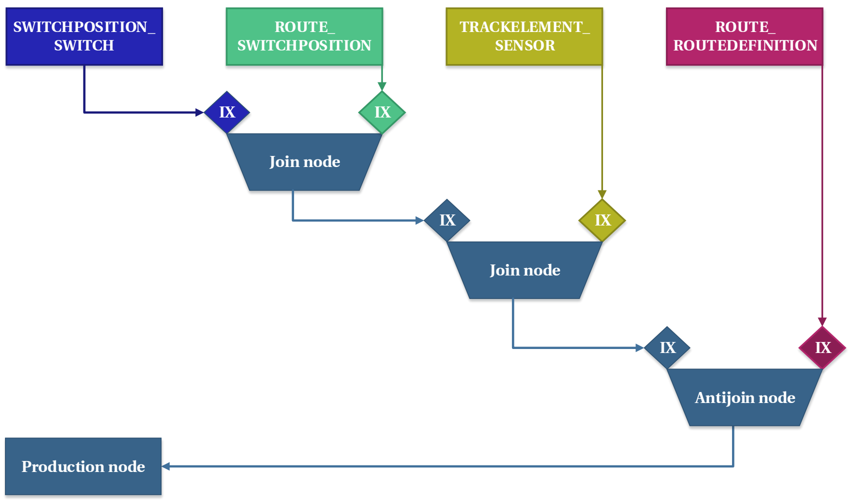

Figure 3.10: *The RouteSensor query's layout*

Deploying the Rete network

The Rete nodes are allocated to the cluster's servers by providing the infrastructure mapping ⑤. In INCQUERY-D's prototype, the Rete recipe's nodes are allocated manually on the cloud servers (called *Machine*s). The Rete nodes are associated with the machines with *infrastructure mapping* relationships. INCQUERY-D's tooling currently provides an Eclipse-based tree editor to define machines and the infrastructure mapping edges.

The tooling is capable of visualizing the Rete network and its mapping to the machines (see Figure 3.11). The Rete network is deployed to the Akka instances running on the servers ⑥.

Evaluating Query

The query is evaluated by initializing the Rete network ⑦ and reading the results from its production node.

Maintaining the Query Results

In order to provide query results that are consistent with the model, we need maintain the Rete network's state. Suppose we have the graph shown on the left side of Figure 3.12 loaded to the Rete network and we decide to delete the `ROUTE_-ROUTEDEFINITION` edge between vertices 2 and 1.

Figure 3.13 shows the distributed Rete network containing the partial matches of the original graph. When we delete the edge between vertices 2 and 1, the `ROUTE_-ROUTEDEFINITION` type indexer receives a notification from the middleware and sends

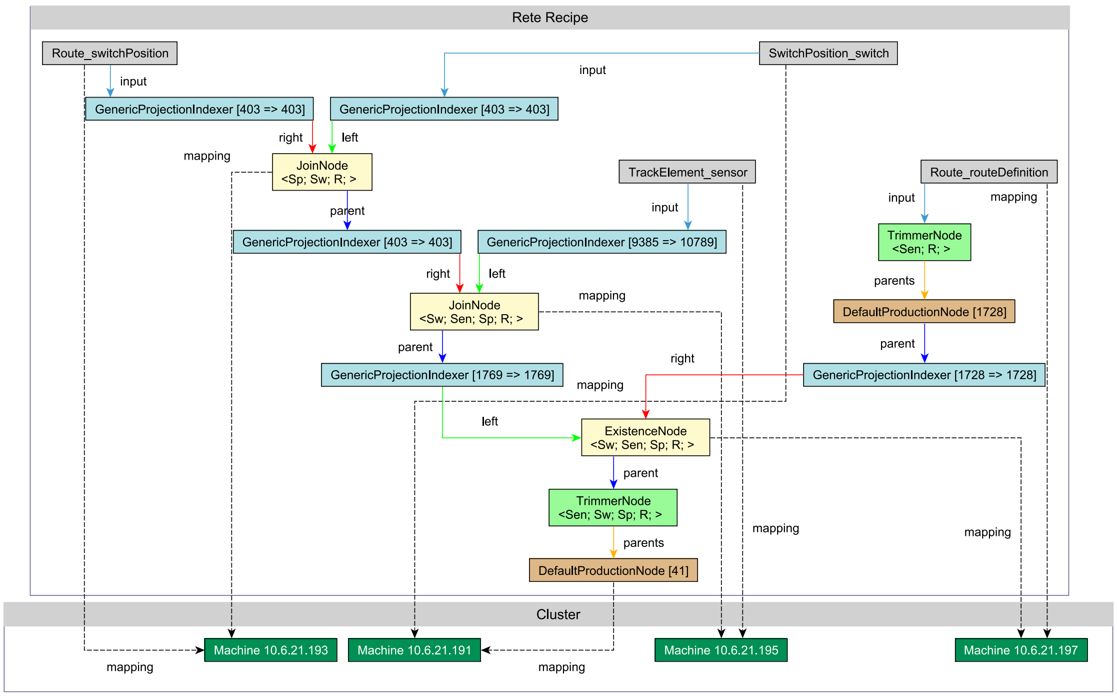

Figure 3.11: *The yFiles viewer in* INCQUERY-D*'s tooling*

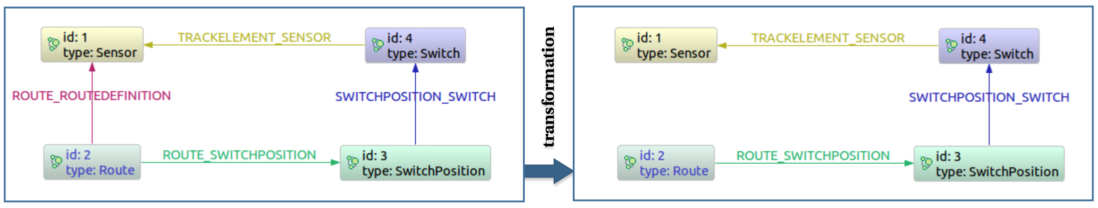

Figure 3.12: *A modification on a Train Benchmark instance model*

a *negative update* ① with the tuple $(2, 1)$. The antijoin node processes the negative update and propagates a negative update ② with the tuple $(3, 4, 2, 1)$. This is received by the production node, which initiates the *termination protocol* ③, ④. After the termination protocol finishes, the indexer signals the client about the successful update. The client can now retrieve the results from the production node. The client may choose to retrieve only the "deltas", i.e. only the the tuples that have been added or deleted since the last modification.

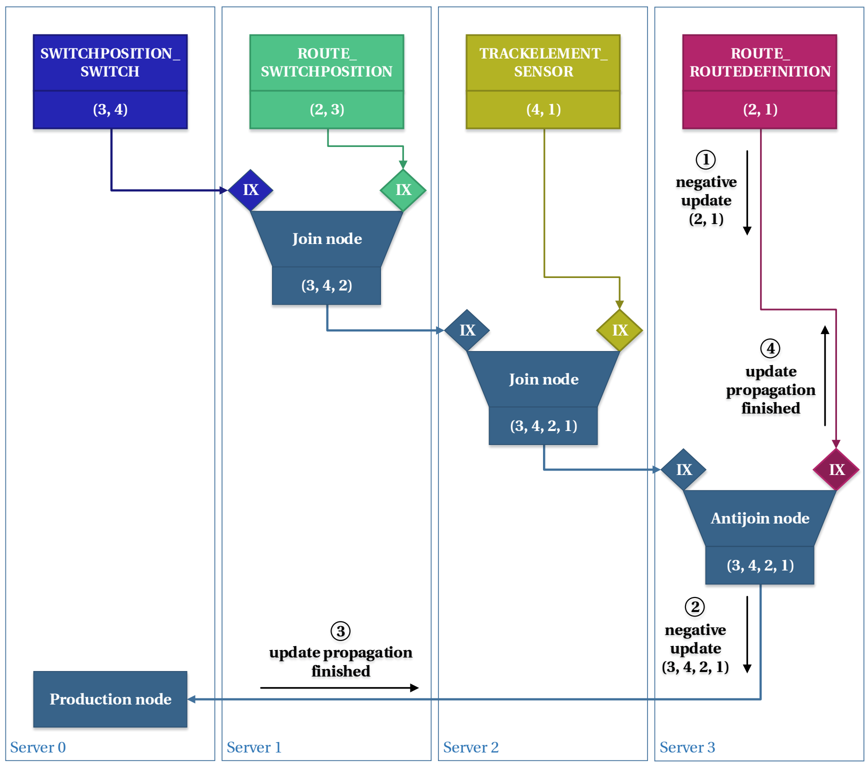

Figure 3.13: *Operation sequence on a distributed Rete network*

Chapter 4

Evaluation of Performance and Scalability

We developed a prototype of INCQUERY-D to evaluate the feasibility of the approach. In the following chapter, based on the work for EMF-INCQUERY, we introduce a distributed performance benchmark. We present the benchmark environment and analyze the results, with particular emphasis on the scalability of our approach.

The prototype of INCQUERY-D is based on the architecture presented in Chapter 3. A working prototype is beneficial for a number of reasons. First, it serves as a proof concept by demonstrating that a distributed, incremental pattern matcher is feasible with the technologies currently available. On the other hand, it gives us the opportunity to define and run benchmarks, so that we can evaluate the scalability aspects of the system.

4.1 Dimensions of Scalability

A distributed system's *scalability* has multiple dimensions. Usually, when aiming for *horizontal scalability*, the most emphasized dimension is the *number of processing nodes* (computers) in the system. However, there are other important aspects that include *local resources* of the servers, *network communication overhead*, etc. The main goals of our benchmark was to measure the scalability of INCQUERY-D with respect to the model size and compare it to other non-incremental query technologies.

4.2 Foundations: the Train Benchmark

The Train Benchmark was designed at the Fault Tolerant Systems Research Group [52, 50] to measure the efficiency of model queries and manipulation operations in different tools. The Train Benchmark is primarily targeted for typical MDE workloads, more specifically for well-formedness validations.

4.2.1 Benchmark Goals

The Train Benchmark measures the response time of the system under test. The benchmark models a ”real-world” MDE workload by simulating a user’s interaction with the model. In this sequence, the user loads the model and validates it against a set queries (defining well-formedness constraints). The user edits the model in small steps. The user’s work is more productive and less error-prone if she receives the results of the validation instantly after each edit. Therefore, the user would like to run re-evaluate well-formedness queries quickly.

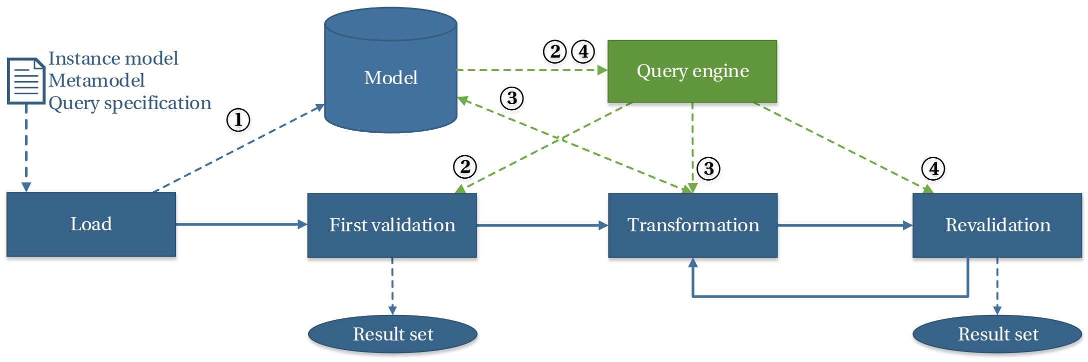

Figure 4.1: *The Train Benchmark’s sequence*

The benchmark defines four distinct phases, also shown on Figure 4.1.

1. *Load:* load the serialized instance model to the database ①.
2. *First validation:* execute the well-formedness query on the model ②.
3. *Transformation:* modify the model ③.
4. *Revalidation:* execute the well-formedness query again ④.

To assess the scalability of the tools, the benchmark uses instance models of growing sizes, each model containing about twice as many model elements as the previous one (Section 4.2.2). Running the same validation sequence on different model sizes highlighted the limitations of the tested query engines.

Scalability is also evaluated against the complexity of the queries. The benchmark defines different queries, each testing different aspects of the query engine (filtering, join and antijoin operations, etc.). To achieve a successful run, the tested tool is expected to evaluate the query and return the *identifier*s of the model elements in the result set.

4.2.2 Generating Instance Models

Due to both confidentiality and technical reasons, it is difficult to obtain real-world industrial models and queries. Also, using confidential data sets hinders the reproducibility of the conducted benchmarks. Therefore, a generator was developed which creates instance models which mimic real-world models.

We used the *railway system metamodel*, defined in Section 3.6. The instance models are generated pseudorandomly, with pre-defined structural constraints and a regular fan-out structure (i.e. nodes of a given type have similar indegree and outdegree) [50]. The generator is capable of generating models of different sizes and formats, including EMF, OWL, RDF and SQL. We also developed a generator for the property graph data model. In Section A.3, we provide some examples about mapping the EMF metamodel to the framework of property graphs.

4.2.3 Original Results for Non-distributed Tools

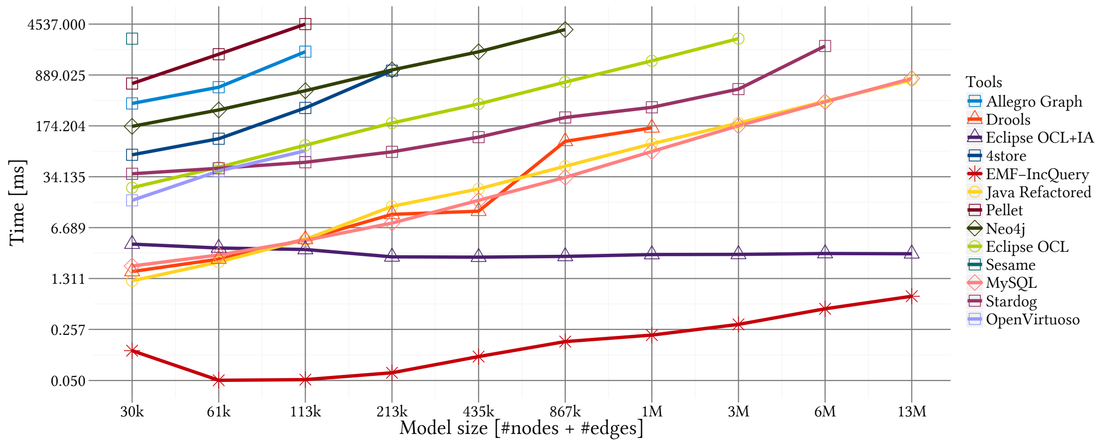

Figure 4.2: *Train Benchmark: reponse times for incremental query evaluation, measured on a single node [12]*

The Train Benchmark was designed to work with different tools originating from various technological spaces, e.g. EMF-based tools (EMF-INCQUERY, Eclipse OCL), semantic web technologies (AllegroGraph, Sesame), NoSQL databases (Neo4j), etc.

Figure 4.2 shows the incremental transformation and validation time for the *Route-Sensor* query, discussed in Section 3.6.1. The results clearly show the advantage of incremental query engines. Both Eclipse OCL Impact Analyzer and EMF-INCQUERY scale very well (their characteristic is almost constant to the model size and linear to the size of the result set), while non-incremental tools scale linearly at best, which renders them inefficient for lange models.

Figure 4.3 shows the memory consumption of the diffent tools. It is apparent that incremental tools space–time tradeoff causes them to consume more memory.

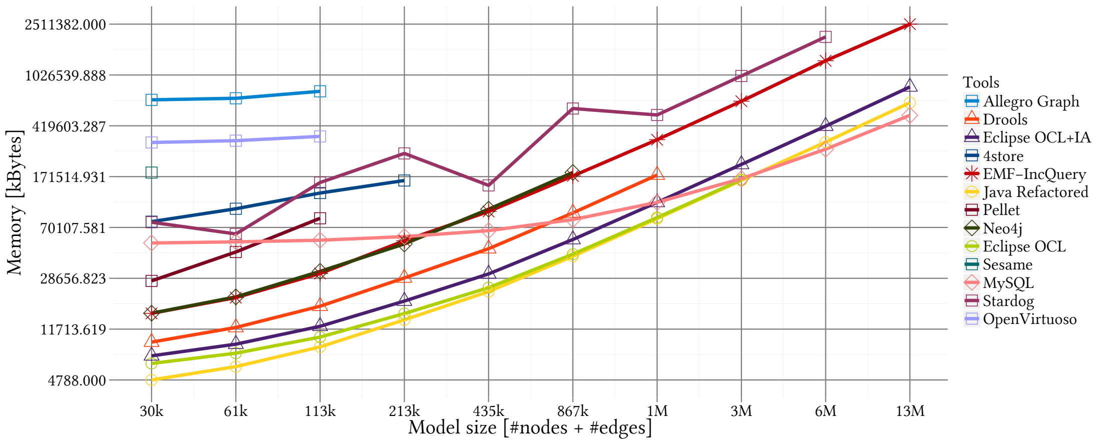

Figure 4.3: *Train Benchmark: memory consumption of the tools [12]*

4.3 Distributed Train Benchmark

Based on the Train Benchmark, discussed in Section 4.2, we created an extended version for distributed systems. The main goal of the distributed Train Benchmark is the same as the original's: measure the reponse time and inspect the scalability of different tools. Specifically, the main goal was to compare INCQUERY-D's performance to distributed, non-incremental query technologies.

4.3.1 Distributed Architecture

The distributed benchmark defines the same phases as the original Train Benchmark (Figure 4.1). The benchmark is controlled by a distinguished node of the system, called the *coordinator*. The coordinator delegates the operations (e.g. loading the graph) to the distributed system. The queries and the model manipulation operations are handled by the database management system which runs them distributedly and waits for the distributed operation to finish (effectively creating a synchronization point after each operation).

4.3.2 Benchmark Limitations

It is possible that the incoming data sets lack a globally unique identifier. In this case, we need to automatically generate unique identifiers. While some systems (e.g. Titan) support this, other systems (e.g. 4store) do not have such feature. For these systems, the INCQUERY-D middleware should be able to generate unique identifiers. This feature is subject to future work (Section 6.3). In the current benchmark, we worked around this by enforcing the generator to create models with numeric unique identifiers[1].

[1]Unlike for property graphs, numeric unique identifiers are not required by the RDF data model.

A common reason for designing and implementing distributed systems is that they are capable of handling a large number of concurrent requests. This way, more users can use the system at the same time. In the distributed Train Benchmark, the system is only used by a single user. Simulating multiple users and issuing concurrent requests is subject to future work (Section 6.3).

4.3.3 Generating Instance Models

For Neo4j, we already expanded the the generator with a *property graph generator* module. The generator creates a graph in a Neo4j database and uses the Blueprints library's `GraphMLWriter` and `GraphSONWriter` classes to serialize it to GraphML (Section A.1.1) and Blueprints GraphSON (Section A.1.2) formats.

Titan's Faunus framework requires a specific format called Faunus GraphSON (Section A.1.3). To use Faunus, we extended the property graph generator to generate Faunus GraphSON files as well.

4.4 Benchmark Environment

We used the distributed Train Benchmark (Section 4.3) to evaluate INCQUERY-D's performance and compare it to non-incremental solutions. In the following section, we will discuss the benchmark setup and the environment in detail.

4.4.1 Benchmark Setup

We tested INCQUERY-D with three storage backends: first with Neo4j, then with 4store (Section 2.3.4) and Titan (Section 2.3.3). In both cases, the system was deployed on a four-node cluster.

As a *non-incremental baseline*, we used Neo4j's and 4store's own query engines. While we also planned to use Titan's query engine, our experiments showed that even for medium-sized graphs, the system was unable to run even the elementary queries (e.g. retrieving vertices by type), not to mention the more complex ones.

The benchmark follows the phases defined in the distributed Train Benchmark. Note that the main difference between the batch and incremental scenarios is that the latter maintain a distributed Rete network, which allows efficient query (re)evaluation.

4.4.2 Hardware and Software Ecosystem

As the testbed, we deployed our system to a private cloud. The cloud is managed by Apache VCL (Virtual Computing Lab) and is also used for educational purposes. Therefore, during the benchmark, the network and the host machines could be under load from other users as well. We consider the effect of these in Section 4.8.

The detailed configuration of the servers are provided below.

Hardware

Each virtual machine used two cores of an Intel Xeon L5420 CPU running at 2.5 GHz and had 8 GBs of RAM. The host machines were connected with gigabit Ethernet network connection.

Software

For the benchmarks, we used the following software stack. The technologies are discussed in Chapter 2.

- Ubuntu 12.10 64-bit
- Oracle Java 7 64-bit
- Neo4j 1.8
- 4store 1.1.5
- Titan 0.3.2
- Faunus 0.3.2
- Hadoop 1.1.2
- Cassandra 1.2.2
- Akka 2.1.2

4.4.3 Benchmark Methodology and Data Processing

Both during the development and in runtime we ensured the *functional equivalence* of the measured tools. *During the development*, we followed the Train Benchmark's well-defined specification [50]. This precisely defines the steps for each phase, e.g. the number of elements to modify in each transformation and the amount of transformation–validation cycles. *In runtime*, we checked the result set for correctness against the reference implementation.

The benchmark coordinator software used the Train Benchmark's framework to collect data about the results of the benchmark. We measured the execution time of the predefined phases. The execution time includes the time required for the coordinator's operation, the computation and IO operations of the cluster's computers and the network communication (to both directions). The execution times were determined using Java's `System.nanoTime()` method.

The results were processed by an R script [27] capable of aggregating and visualizing the results.

4.5 Benchmark Results with Neo4j

During the earlier phases of the research, we conducted measurement using only Neo4j. These results were published in [51]. The benchmark's setup was slightly different, with the main difference being that due to the lack of sharding in Neo4j, we *sharded the graph manually*. This had some important implications.

- The *batch* queries were ran on all shards separately and their results were aggregated by the coordinator. The transformations also ran separately.
- The *incremental* queries were evaluated with a distributed Rete network. The elementary model queries (for filling the indexers) were ran on all shards separately and aggregated by the indexers. The transformations also ran separately.

Because the graph was sharded to disjoint partitions with no edges between them, this can be viewed as an ideal case of graph sharding. Therefore, we can use the results to inspect the an "ideal" sharding strategy's impact on the performance. We present the most important results of the benchmark.

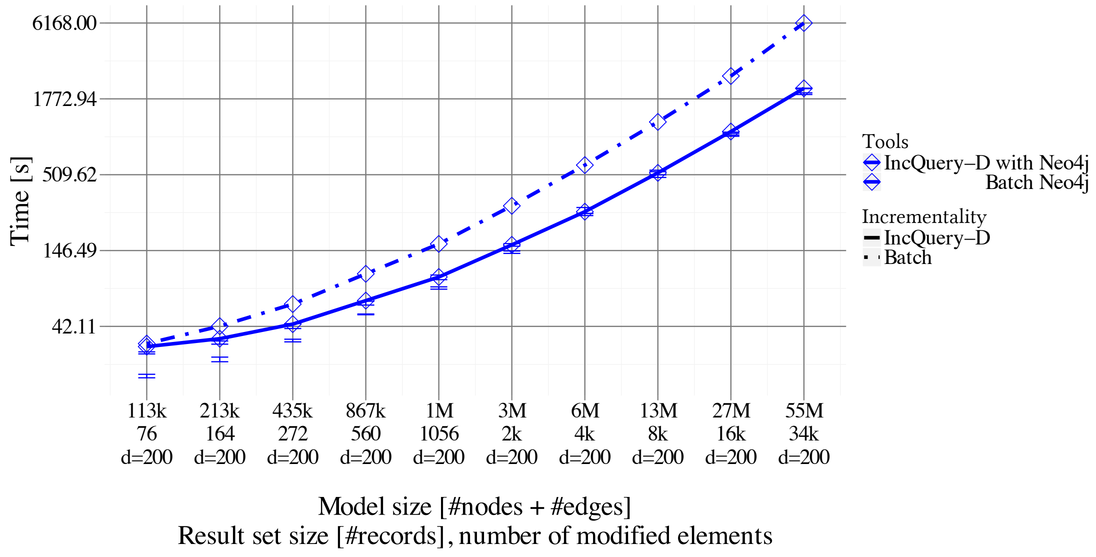

Figure 4.4: *Total execution times for 50 validations*

Figure 4.4 shows that INCQUERY-D with Neo4j consistently outperforms Neo4j's query engine.

Figure 4.5 shows that for transformation and revaliation, INCQUERY-D with Neo4j is about two orders of magnitude faster than Neo4j's query engine.

4.6 Benchmark Results with 4store and Titan

This section presents the benchmark results with 4store and Titan. Unlike the benchmark with Neo4j (Section 4.5), this benchmark used truly distributed storage backends.

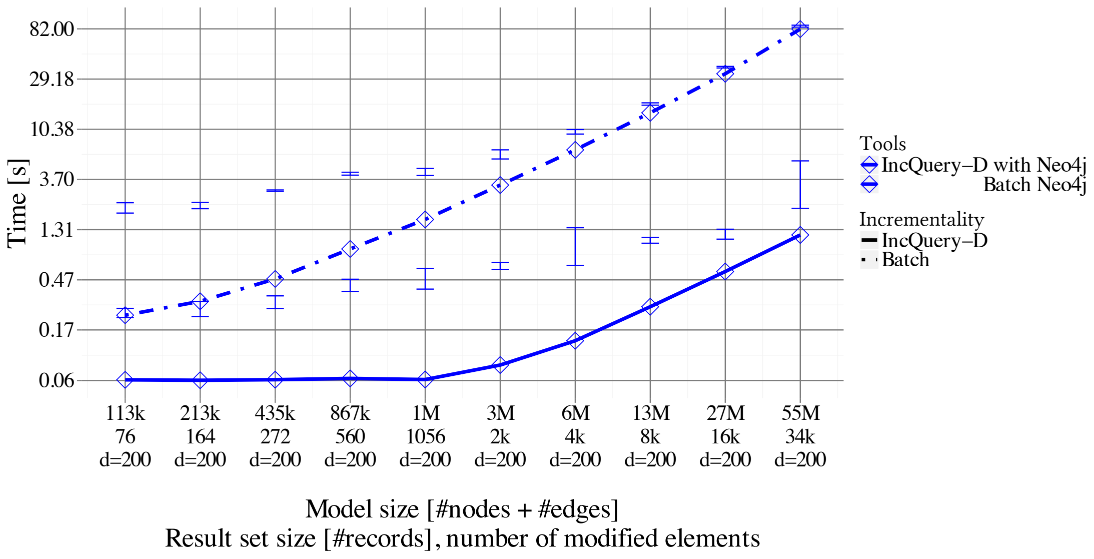

Figure 4.5: *Execution times for transformation and revalidation*

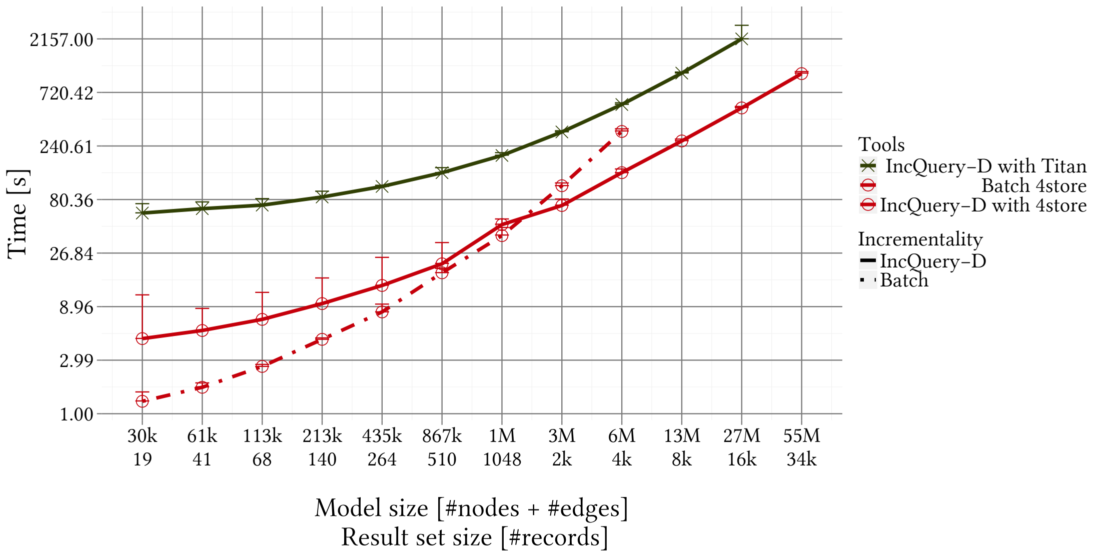

Figure 4.6: *Execution times for load and first validation*

The execution times for the *load and first validation* phases are shown on Figure 4.6. As expected, due to the overhead of the Rete network's construction, the *batch* tool is faster for small models. However, it is important to observe that even for medium-sized models (with a couple of million elements), the INCQUERY-D tools start to edge ahead. This shows that the Rete network's construction overhead already pays off for the first validation.

The execution times for the *transformation* phase are shown on Figure 4.7. The incremental tools provide faster transformation times due to the fact that instead of querying the database, the modeling application can rely on the query layer's indexers. Even for medium-sized models, the INCQUERY-D tools are more than two orders

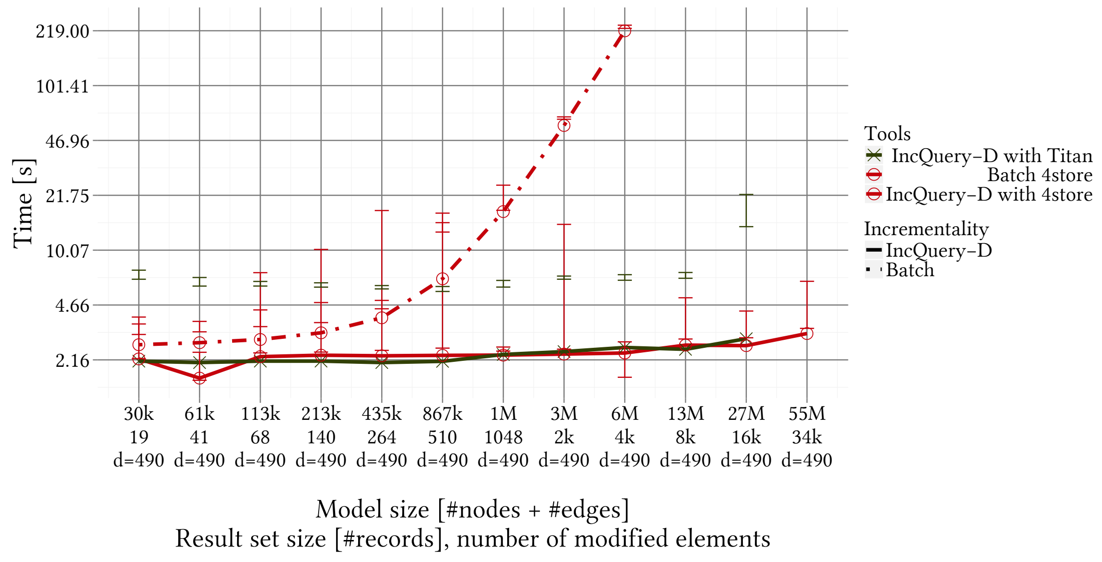

Figure 4.7: *Execution times for transformation*

of magnitude faster than the batch tool.

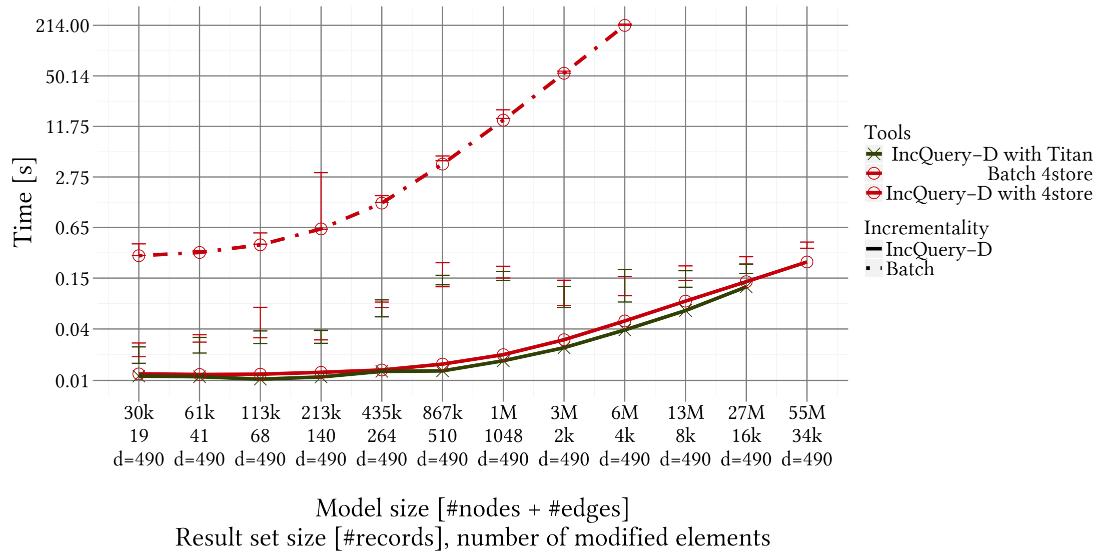

Figure 4.8: *Execution times of the revalidation*

The incremental tools have an even greater advantage for *revalidation* times, shown on Figure 4.8. For medium-sized models, they are more than three orders of magnitude faster than the batch tool.

This shows that INCQUERY-D is not just capable of processing models with tens of millions of elements (well beyond the capabilities of single-node tools), but also, it provides sub-second revalidation times.

Figure 4.9 shows the total execution time for a sequence: loading the model, then running transformations and revalidations 50 times. Due to the large number of transformations and revalidations, incremental tools are significantly faster. For example,

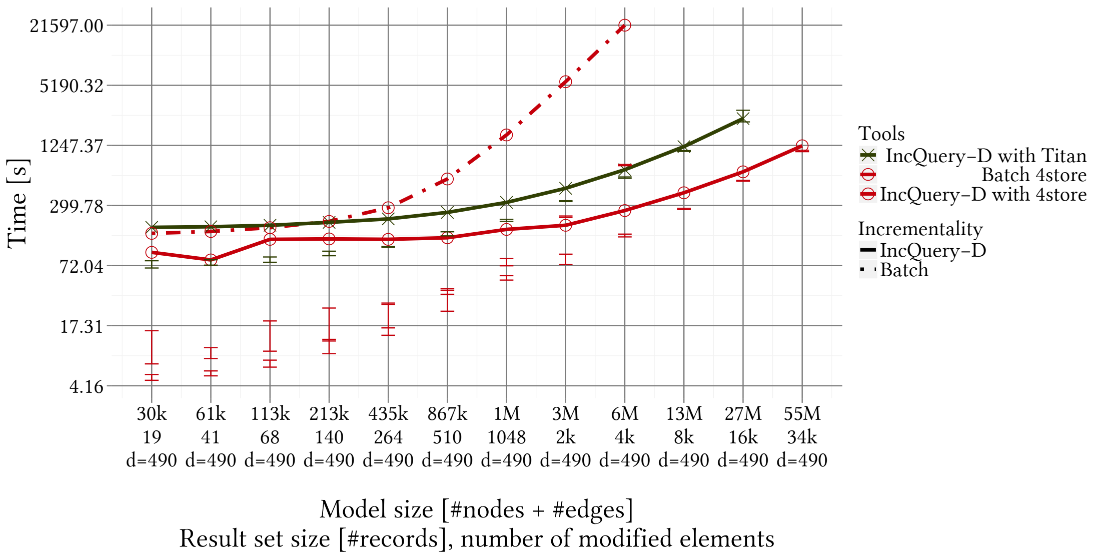

Figure 4.9: *Total execution times for 50 validations*

for a model with 6 million elements, the batch tool took almost 6 hours, while the 4store-based incremental tool took less than 5 minutes.

4.7 Result Analysis

The results clearly show that the initialization of the Rete network adds some overhead during the *load and first validation phases*. However, even for medium-sized models, this is easily outweighed by the high query performance of the Rete network.

The almost constant characteristic of the execution times of the INCQUERY-D tools' *transformation* and *validation* phases confirm that a distributed, scalable, incremental pattern matcher is feasible with current technologies. Based on the results, we can conclude that while network latency is present, the distributed Rete network still allows sub-second on-the-fly model validation operations. It is also important to observe the similar characteristic of INCQUERY-D's and EMF-INCQUERY's transformation and validation times (Figure 4.2).

Another important observation is that for INCQUERY-D tools, the execution time is approximately proportional to *the size of the change*. For batch tools, it is proportional to the *size of the model*.

Note that these results and scalability characteristics do not apply for every workload profile. For example, if the user modifies large chunks of the model and issues queries infrequently, batch query evaluation methods can be faster.

The high memory consumption of the Rete algorithm was one of our main motivations to build a distributed system. For very large models (beyond 10^8 model elements, we ran into cases where the Java Virtual Machine ran out of or had just enough memory. This resulted in `OutOfMemoryError: Java heap space` and `OutOfMemoryError: GC overhead limit exceeded` exceptions, respectively. Introducing a *Rete node sharding* or other fault-tolerance mechanisms for these cases is subject to future work (Section 6.3).

The results show that the 4store-based INCQUERY-D prototype is consistently faster in the *load* phase than the Titan-based one. This is due to 4store's simpler architecture and different data model, which is better suited to the INCQUERY-D middleware's elementary model queries.

In accordance with the original Train Benchmark's results, the distributed Train Benchmark proved that incremental tools have an advantage for transformation and well-formedness validation sequences. Compared to the Train Benchmark, we managed to work with significantly larger models with more than 50 million model elements. Based on the results, we expect INCQUERY-D to also perform well on different data sets and queries.

4.8 Threats to Validity

To guarantee the correctness of our benchmarks, we laid out some rules to ensure the precision of the results.

First, to start each benchmark sequence independently, we turned the operating system's caching mechanisms off. The execution time of the *validation and transformation phases* were determined by running them 50 times and taking the *median* values (we decided to take the median instead of the mean value, because the former is less sensitive to transient effects). This way, we could measure the Java Virtual Machine's warmup effect, which would also occur in a real-world model query engine running for several hours or even longer.

As discussed in Section 4.4.2, our servers could be influenced from the workload caused by other users of the same cloud. To minimalize the effect of this and other transient loads, we ran the benchmark five times and took the *minimum* value for each phase. We also disabled file caching in the operating system, so that the serialized model always must be read from the disk.

Despite our efforts, transient effects could still be present in the results. However, their effect is only a threat for smaller model sizes, where the measured execution times are low. For larger models, which are the main targets of our work, due to longer execution times, the transient effects do not threat the validity of the benchmark results.

4.9 Summary

Our benchmarks proved that the proposed architecture is capable of providing scalable, incremental query evaluation. INCQUERY-D's scalability characteristics confirmed that despite the additional network latency, it is possible to keep EMF-INCQUERY's almost constant performance characteristics in a distributed environment. The results show the model size barrier, primarily caused by limitations of memory, can be pushed further using a horizontal scaling approach.

It is important to note that our benchmark did not cover all aspects of distributed scalability. For example, simulating multiple users, measuring the exact memory consumption and network traffic of each server is subject to future work.

Chapter 5

Related Work

A wide range of special languages have been developed to support *graph-based* representation and querying of computer data. This chapter collects the research and development works that are related to INCQUERY-D.

5.1 Eclipse-based Tools

A class-diagram like modeling language is Ecore of the EMF (Eclipse Modeling Framework, discussed in Section 2.3.1), 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 (Object Constraint Language) is a declarative constraint description and query language that can be evaluated with the local-search based Eclipse OCL [37] engine. To address scalability issues, *incremental* impact analysis tools [43] have been developed as extensions or alternatives to Eclipse OCL.

5.2 Rete Implementations

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 [63] uses this algorithm to perform complex event processing (formulated in SPARQL) on RDF data, gathered from distributed sensors.

Diamond [58] uses a *distributed Rete network* to evaluate 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 [34], 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 *dis-*

tributed incremental query evaluation in the context of model-driven engineering. As the architecture of IncQuery-D separates the data store from the query engine, we believe that the scalable processing of RDF and 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 [29]. 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.

5.3 Benchmarking

In the research work undertaken in the Budapest University of Technology and Economics, numerous benchmarks were designed and elaborated for graph pattern matching and graph transformation [70, 48].

The distributed Train Benchmark used in this report builds on the most recent results, published in [50].

Chapter 6

Conclusions

This chapter summarizes the contributions presented in the report.

6.1 Summary of Contributions

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. The feasibility of the approach 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 50 million elements.

During the research and development of INCQUERY-D so far, I achieved the following results.

6.1.1 Scientific Contributions

I achieved the following scientific contributions:

- I proposed a novel architecture for building a distributed, scalable, incremental graph query engine over different storage backends. The architecture was published in [49].

- I designed and implemented a *distributed, asynchronous version of the Rete algorithm.*

- I extended the termination protocol used EMF-INCQUERY to work in a distributed environment.

- I extended the Train Benchmark to work in a distributed environment.

- 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.*

6.1.2 Practical Accomplishments

I achieved the following practical accomplishements:

- 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.
- I extended the Train Benchmark with a *new instance model generator*, which can produce property graphs and serialize them in various formats: GraphML, Blueprints GraphSON and Faunus GraphSON.
- I developed INCQUERY-D's prototype, including the query layer, the middleware and the integration to different storage technologies. I wrote more than 3000 lines of Java code and approximately 500 lines of configuration and deployment scripts.
- I elaborated automated deployment tools based on EMF-INCQUERY's existing technologies.
- I experimented with modern non-relational database management systems with a focus on NoSQL graph databases and triple stores. For the purpose of benchmarking different tools, I created scripts to install various graph storages.
  - I deployed a manually sharded *Neo4j* cluster. I formulated the appropriate Cypher queries and created the connector class in INCQUERY-D's middleware to access Neo4j.
  - 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 and the Faunus graph analytics engine. I formulated the necessary the Gremlin queries and created the connector class in INCQUERY-D's middleware.
  - I implemented scripts to install the *4store triplestore* on a cluster. I formulated the necessary the SPARQL queries and created the connector class in INCQUERY-D's middleware.

- I implemented scripts for *automating the benchmark* and *operating a cluster of Akka microkernels*.

6.2 Limitations

INCQUERY-D's current implementation has some limitations, the most important ones are the following.

1. The Rete nodes are allocated manually. The user has to define the mapping between the Rete network and the infrastructure. However, given a mapping, the system is capable of automatically deploying the Rete network.
2. Only a subset of the nodes defined in the Rete algorithm are implemented. For example, the current implementation does not support recursive patterns, checking property conditions and transitive closures.
3. The Eclipse-based tooling does not cover the whole workflow. The user is required to do some manual work, e.g. running scripts manually.

6.3 Future work

For future work, we plan to address the aforementioned limitations.

1. The allocation of the Rete nodes will be supported using techniques like CSP (Constraint Satisfaction Problem) solvers and DSE (Design Space Exploration) [45]. We plan to further explore advanced optimization challenges such as dynamic reconfiguration and fault tolerance.
2. We will complete the implementation of the nodes defined in Rete algorithm.
3. The tooling in under active development with plans for a *live monitoring* feature.

We also plan to extend the distributed Train Benchmark to model different real-world workloads, e.g. simulating multiple users issuing concurrent requests.

Another direction is experimenting with programming languages that are better suited to asynchronous algorithms, e.g. Erlang and Scala, a Java-based functional object-oriented programming language. For our storage layer, we plan to test distributed in-memory databases, e.g. Hazelcast [15]. Also, we are constantly looking for alternative scalable persistent graph database technologies.

List of Figures

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. `http://4store.org/`, October 2013.

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

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

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

[8] Apache HBase. `http://hbase.apache.org/`, May 2013.

[9] Apache Thrift. `http://thrift.apache.org/`, October 2013.

[10] Avahi. `http://www.avahi.org/`, October 2013.

[11] Blueprints. `http://blueprints.tinkerpop.com/`, May 2013.

[12] Efficient Instance-level Model Validation by Incremental Query Techniques - preliminary. `http://incquery.net/publications/trainbenchmark/full-results`, October 2013.

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

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

[15] In-Memory Data Grid – Hazelcast. `http://www.hazelcast.com/index.jsp`, October 2013.

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

[17] MongoDB. `http://www.mongodb.org/`, October 2013.

[18] Neoclipse. `https://github.com/neo4j/neoclipse`, May 2013.

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

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

[21] OrientDB Graph-Document NoSQL DBMS. `http://www.orientdb.org/`, October 2013.

[22] Planet Cassandra – Companies. `http://planetcassandra.org/Company/ViewCompany`, October 2013.

[23] Protocol Buffers – Google's data interchange format. `https://code.google.com/p/protobuf/`, October 2013.

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

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

[26] The Innovative Zing JVM. `http://www.azulsystems.com/sites/default/files/images/Innovative_Zing_JVM_v2.pdf`, August 2013.

[27] The R Project for Statistical Computing. `http://www.r-project.org/`, October 2013.

[28] TinkerPop. `http://www.tinkerpop.com/`, May 2013.

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

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

[31] 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.

[32] Gábor Bergmann. *Incremental Model Queries in Model-Driven Design.* Ph.D. dissertation, Budapest University of Technology and Economics, Budapest, 10/2013 2013.

[33] Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró. *Incremental Evaluation of Model Queries over EMF Models: A Tutorial on EMF-IncQuery*, volume 6698 of *Lecture Notes in Computer Science*, pages 389–390. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-21470-7_32.

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

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

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

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

[38] Eclipsepedia. CDO. `http://wiki.eclipse.org/CDO`, October 2013.

[39] EMF documentation. Package org.eclipse.emf.ecore. `http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html`, October 2013.

[40] Andreas Emil Feldmann and Luca Foschini. Balanced partitions of trees and applications. In Christoph Dürr and Thomas Wilke, editors, *STACS*, volume 14 of *LIPIcs*, pages 100–111. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

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

[42] 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.

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

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

[45] 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%.

[46] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[47] 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.

[48] Ákos Horváth, Gábor Bergmann, István Ráth, and Dániel Varró. Experimental assessment of combining pattern matching strategies with viatra2. *International Journal on Software Tools for Technology Transfer*, 12(3-4):211–230, 2010.

[49] Benedek Izsó, Gábor Szárnyas, István Ráth, and Dániel Varró. IncQuery-D: incremental graph search in the cloud. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 4:1–4:4, New York, NY, USA, 2013. ACM.

[50] Benedek Izsó, Zoltán Szatmári, Gábor Bergmann, Ákos Horváth, and István Ráth. Towards Precise Metrics for Predicting Graph Query Performance. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*, 2013. Accepted.

[51] 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.

[52] Benedek Izsó, Zoltán Szatmári, and István Ráth. High performance queries and their novel applications, 2012.

[53] Adam Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):36–44, August 2009.

[54] Jim Webber. On Sharding Graph Databases. `http://jim.webber.name/2011/02/on-sharding-graph-databases/`, February 2011.

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

[56] Michael G. Noll. Running Hadoop on Ubuntu Linux (Multi-Node Cluster). `http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/`, October 2013.

[57] 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.

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

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

[60] Javier Espinazo Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. Morsa: a scalable approach for persisting and accessing large models. In *Proceedings of the 14th international conference on Model driven engineering languages and systems*, MODELS'11, pages 77–92, Berlin, Heidelberg, 2011. Springer-Verlag.

[61] Redland RDF Libraries. Raptor RDF Syntax Library. `http://librdf.org/raptor/`, October 2013.

[62] Redland RDF Libraries. Rasqal RDF Query Library. `http://librdf.org/rasqal/`, October 2013.

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

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

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

[66] Markus Scheidgen, Anatolij Zubow, Joachim Fischer, and Thomas H. Kolbe. Automated and transparent model fragmentation for persisting large models. In *Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems*, MODELS'12, pages 102–118, Berlin, Heidelberg, 2012. Springer-Verlag.

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

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

[69] The MOGENTES project. Model-Based Generation of Tests for Dependable Embedded Systems. `http://www.mogentes.eu/`.

[70] Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for graph transformation. In *Proc. of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 79–88, Dallas, Texas, USA, September 2005.

Appendix A

Graph Formats

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

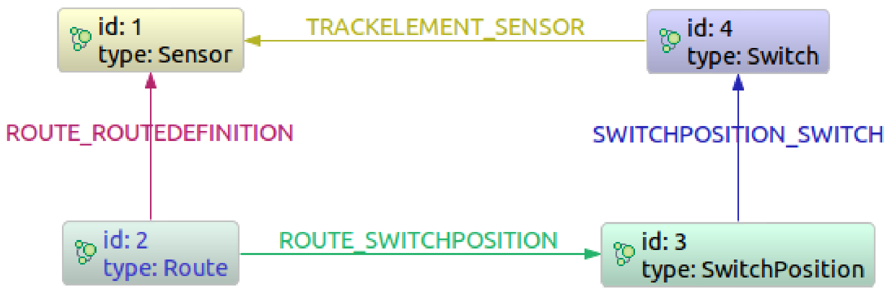

Figure A.1: *An example graph based on the railway system metamodel*

A.1 Property Graph Formats

A.1.1 GraphML

The GraphML format [25] is the most widely used graph representation format, based on XML (Extensible Markup Language). It has strong tooling support between graph databases and graph visualizing tools.

```
<?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 railway system metamodel stored in GraphML format*

A.1.2 Blueprints GraphSON

Blueprints GraphSON [14] is a JSON-based (JavaScript Object Notation) format. It is not as well supported as the GraphML format (Section A.1.1), but it is less verbose and more readable.

```
{
  "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 railway system metamodel stored in Blueprints GraphSON format*

A.1.3 Faunus GraphSON

In the Faunus GraphSON format [13], each line is a separate JSON (JavaScript Object Notation) document representing a vertex in the graph. This way, the file can be splitted to blocks efficiently and processed on Hadoop nodes in a parallel way.

```
{"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 railway system metamodel stored in Faunus GraphSON format*

A.2 Semantic Graph Formats

A.2.1 RDF/XML

RDF/XML is an XML-based (Extensible Markup Language) format for serializing RDF triples.

```
<?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 railway system metamodel stored in RDF format*

A.3 Mapping Ecore to Property Graphs

Mapping the Ecore kernel's concepts to property graphs is not a trivial task. We developed the property graph generator module for the Train Benchmark based on the railroad system's Ecore metamodel (Section 3.6.1), which meant the Ecore concepts had to be mapped to property graphs. Following the mapping defined in Section 2.2.2, we created the equivalent instance models for property graphs as well. Below, we provide some examples about the mapping:

- `Segment` is an `EClass` instance. In a property graph, types cannot be represented explicitly. Instead, for each node representing a `Segment` instance, we add a `type` property with the value `Segment`.
- `Segment_length` is an `EAttribute` instance. For each graph node representing a `Segment`, we define a property with the value `Segment_length`.
- `TrackElement_Sensor` is an `EReference` instance. For each edge representing a `TrackElement_Sensor` instance, we add the `TRACKELEMENT_SENSOR` label.
- `EInt` in an `EDataType` instance. Each attribute with this type, e.g. the `Sensor` class' `Segment_length` attribute, is defined with the Java primitive type `int`.