



**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 24, 2013



# Kivonat

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

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

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

# Abstract

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

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

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

# Contents

<b>1</b>	<b>Introduction</b>	<b>8</b>
1.1	Context . . . . .	8
1.2	Problem Statement and Requirements . . . . .	9
1.3	Objectives . . . . .	9
1.4	Contribution, Added Value . . . . .	9
1.5	Structure of the Report . . . . .	9
<b>2</b>	<b>Background Technologies</b>	<b>10</b>
2.1	Big Data and the NoSQL Movement . . . . .	10
2.2	Graph Models and Storage Methods . . . . .	11
2.2.1	TinkerPop . . . . .	12
2.2.2	Triplestores . . . . .	13
2.3	NoSQL Technologies . . . . .	13
2.3.1	Cassandra . . . . .	14
2.3.2	Hadoop . . . . .	15
2.3.3	Neo4j . . . . .	17
2.4	Graph Technologies . . . . .	17
2.4.1	Titan . . . . .	17
2.4.2	4store . . . . .	19
2.5	Asynchronous Messaging . . . . .	19
2.5.1	Akka . . . . .	19
2.6	Eclipse-based Technologies . . . . .	20
2.6.1	EMF . . . . .	21
2.6.2	EMF-INCQUERY . . . . .	21
2.7	Challenges . . . . .	23
<b>3</b>	<b>Overview of the Approach</b>	<b>24</b>
3.1	Architecture of Incremental Queries: the Single Node Case . . . . .	24
3.1.1	Incremental Query Evaluation . . . . .	24
3.1.2	Rete in EMF-INCQUERY . . . . .	26

3.2	Extensions for Distributed Scalability . . . . .	28
3.2.1	INCQUERY-D's Architecture . . . . .	29
3.2.2	Indexing and Initialization . . . . .	29
3.2.3	Distributed Storage Layer . . . . .	29
3.2.4	Notification in Distributed Database Management Systems . . . . .	30
3.2.5	Graph-like Data Manipulation . . . . .	30
3.2.6	Degrees of Freedom . . . . .	31
3.2.7	Distributed Termination Protocol . . . . .	31
3.2.8	Workflow . . . . .	32
3.3	Tooling . . . . .	32
3.3.1	Mapping Ecore to Other Data Models . . . . .	33
3.3.2	Runtime Model-based Dashboard . . . . .	33
3.4	Elaboration of the Example . . . . .	34
3.4.1	Case Study: Railroad System Design . . . . .	35
3.4.2	Workflow of the Example . . . . .	37
<b>4</b>	<b>Evaluation of Scalability and Performance</b>	<b>41</b>
4.1	Goals . . . . .	41
4.1.1	Dimensions of Scalability . . . . .	41
4.2	TrainBenchmark . . . . .	42
4.2.1	Benchmark Goals . . . . .	42
4.2.2	Results . . . . .	42
4.2.3	Generating Models . . . . .	43
4.3	Distributed TrainBenchmark . . . . .	43
4.3.1	Generating Models . . . . .	43
4.3.2	Distributed Architecture . . . . .	44
4.3.3	Benchmark Limitations . . . . .	44
4.4	Benchmark Environment . . . . .	44
4.4.1	Benchmark Setup . . . . .	44
4.4.2	Hardware and Software Ecosystem . . . . .	45
4.4.3	Benchmark Methodology and Data Processing . . . . .	46
4.5	Results . . . . .	47
4.5.1	Result visualizations . . . . .	47
4.6	Result Analysis . . . . .	49
4.6.1	Memory Consumption . . . . .	49
4.6.2	Threats to Validity . . . . .	50
4.7	Benchmark Results with Neo4j . . . . .	50
4.8	Summary . . . . .	51

<b>5</b>	<b>Related work</b>	<b>52</b>
5.1	Eclipse-based Tools . . . . .	52
5.2	Semantic Web and NoSQL . . . . .	52
5.3	Rete Implementations . . . . .	53
<b>6</b>	<b>Conclusions</b>	<b>54</b>
6.1	Summary of Contributions . . . . .	54
6.1.1	Scientific Contributions . . . . .	54
6.1.2	Practical Accomplishments . . . . .	55
6.2	Limitations . . . . .	55
6.3	Future work . . . . .	55
	<b>Bibliography</b>	<b>58</b>
<b>A</b>	<b>Graph Formats</b>	<b>64</b>
A.1	Property Graph Formats . . . . .	64
A.1.1	GraphML . . . . .	64
A.1.2	Blueprints GraphSON . . . . .	65
A.1.3	Faunus GraphSON . . . . .	66
A.2	Semantic Graph Formats . . . . .	66
A.2.1	RDF/XML . . . . .	66

# Chapter 1

## Introduction

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

### 1.1 Context

Many scalability issues can be addressed by improving query performance. *Incremental evaluation* of model queries aims to reduce query response time by limiting the impact of model modifications to query result calculation. Such algorithms work by either (i) building a cache of interim query results and keeping it up-to-date as models change (e.g. EMF-INCQUERY [31]) 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 [39]). 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 [60]). 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 [54, 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 [40]) has crucial conceptual and technological differences to traditional meta-modeling languages such as Ecore [60]. Additionally, while there are initial efforts to overcome the mapping issues between the MDE and Linked Data worlds [44], even the most sophisticated NoSQL storage technologies lack efficient and mature support for executing expressive queries incrementally.

## **1.2 Problem Statement and Requirements**

[57]

## **1.3 Objectives**

## **1.4 Contribution, Added Value**

The main contributions of this report are a novel architecture for distributed, incremental query engine and the evaluation of the architecture on a working prototype.

## **1.5 Structure of the Report**

The report is structured as follows. Chapter 2 introduces the background technologies and the motivation behind 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

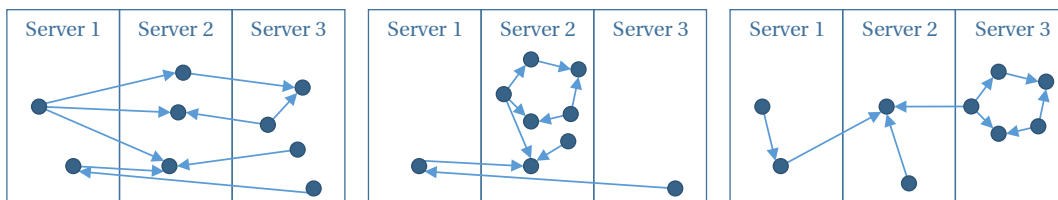
## Background Technologies

Implementing a scalable graph pattern matcher requires a wide range of technologies. Careful selection of the technologies is critical to the project’s success. For INCQUERY-D, we looked for technologies that can form the building blocks of a distributed, scalable model repository and pattern matcher. These technologies must be designed with scalability in mind and deployed at scale successfully. To avoid licensing issues, our search criteria included that, if possible, the technologies should be free and open-source solutions.

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 technologies that can form the basis of a scalable, distributed, asynchronous system. We also present EMF (Eclipse Modeling Framework), and EMF-INCQUERY, an incremental model query evaluation tool.



**Figure 2.1:** *Different sharding strategies for the same graph*

## 2.1 Big Data and the NoSQL Movement

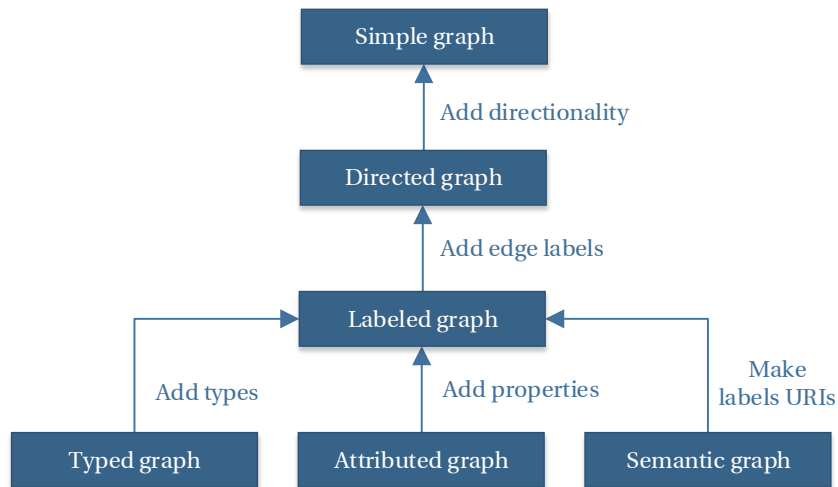
Since the 1980s, database management systems based on the relational data model [33] dominated the database market. Relational databases have a number of important advantages: precise mathematical background, understandability, mature tooling and so on. However, due to their rich feature set and the strongly connected nature of their data model, relational databases often have scalability issues [49, 58]. In practice, this renders 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*<sup>1</sup> [17]. During the development of INCQUERY-D’s prototype, we experimented with numerous NoSQL databases.

## 2.2 Graph Models and Storage Methods

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.



**Figure 2.2:** Different graph data models (based on [56])

<sup>1</sup>The community now mostly interprets NoSQL as “not only SQL”.

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*).

*RDF graphs* use URIs (Uniform Resource Identifiers) instead of labels, otherwise they have similar expressive power as labeled graphs. *Property graphs* add even more possibilities 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`.

*Typed graphs*

*Attributed graphs*

*Semantic graphs*

Property graphs are powerful enough to describe Java objects or EMF instance models.

## 2.2.1 TinkerPop

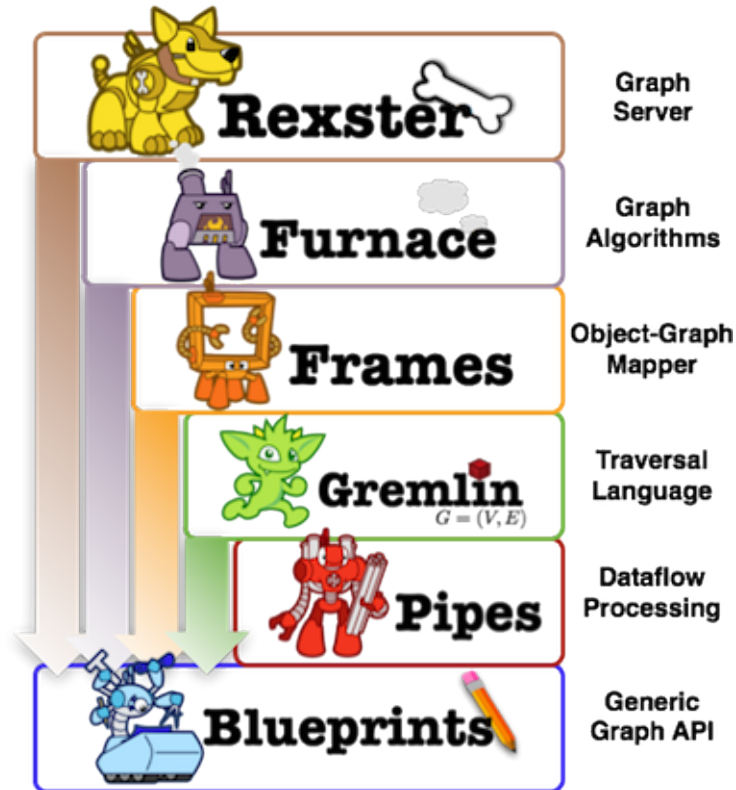
The *TinkerPop* framework is an open-source software stack for graph storage and processing [26]. TinkerPop includes *Blueprints*, a property graph model interface. Blueprints fulfills the same role for graph databases as JDBC does for relational databases. Most NoSQL graph databases implement the property graph interface provided by Blueprints, including Neo4j (Section 2.3.3), Titan (Section 2.4.1), DEX [22], InfiniteGraph [18] and OrientDB [19].

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.

```
1 gremlin> g.V('name', 'John').out('father')
2 ==>Jack
3 gremlin> g.V('name', 'John').out('father').out('father')
4 ==>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.



**Figure 2.3:** The TinkerPop software stack [12]

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

## 2.2.2 Triplestores

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

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

## 2.3 NoSQL Technologies

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

### 2.3.1 Cassandra

Cassandra is one of the most widely used NoSQL databases [9]. Originally developed by Facebook [50], Cassandra is now an open-source Apache project. The whole project is written in Java.

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

Cassandra has sophisticated 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 [20].

### Consistent Hashing

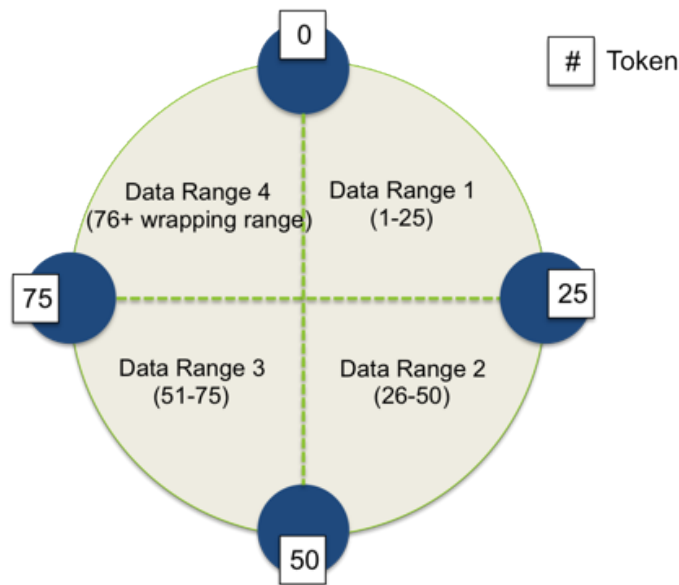
To distribute the data across the cluster, Cassandra uses a partitioner mechanism. The basic partitioners distribute the rows evenly based on their key's hash value. In a cluster with  $n$  nodes, the naïve method for determining the location for a row with key  $x$  is computing  $h(x) \bmod n$ , where  $h(x)$  is the hash function. Currently, Cassandra provides partitioners based on the MD5 and the Murmur3 hash function. However, this approach has a serious limitation: if we remove or add nodes to the cluster, we have to recompute the hash values and possibly relocate almost all rows in the cluster. To avoid this, Cassandra uses a special kind of hashing called *consistent hashing*.

The ring is divided into ranges equal to the number of nodes, with each node being responsible for one or more ranges of the data. Before a node can join the ring, it must be assigned a *token*. The token value determines the node's position in the ring and its range of data. The ring is walked clockwise until it locates the node with a token value greater than that of the row key. With the nodes sorted in token order, the last node is considered the predecessor of the first node; hence the ring representation<sup>2</sup> [7].

---

<sup>2</sup>Note "consistent" here is different from both the idea of consistency in data consistency and in the ACID (atomicity, consistency, isolation, durability) properties guaranteed by transactions. It refers to the

For example, consider a simple four-node cluster (Figure 2.3), where all of the row keys managed by the cluster are in the range of 0 to 100. Each node is assigned a token that represents a point in this range. In this example, the token values are 0, 25, 50 and 75. The first node (with token 0), is responsible for the wrapping range (76+), the second node (with token 25) is responsible for the data range 1 – 25, and so on [7].



**Figure 2.4:** *Cassandra's ring for data partitioning [7]*

### 2.3.2 Hadoop

Hadoop is an open-source, distributed data processing framework inspired by Google's publications about MapReduce [34] and the Google File System [38]. Originally developed at Yahoo!, Hadoop is now an Apache project [10]. Like Google's systems, Hadoop is designed to run on commodity hardware, i.e. server clusters built from commercial off-the-shelf products. Hadoop provides a distributed file system (HDFS) and a column family database (HBase). All software in the Hadoop framework is written in Java.

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

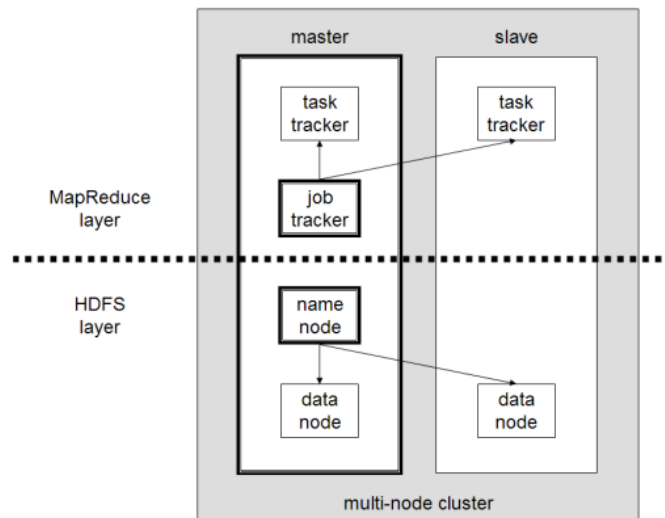
Hadoop is often used for sorting, filtering and aggregating data sets. It is also used for fault-tolerant, distributed task execution.

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

---

fact that tries to map the same rows to the same machine, even if the number of machines ( $n$ ) changes over time slightly.

the slave nodes' *task tracker* modules (Figure 2.4).

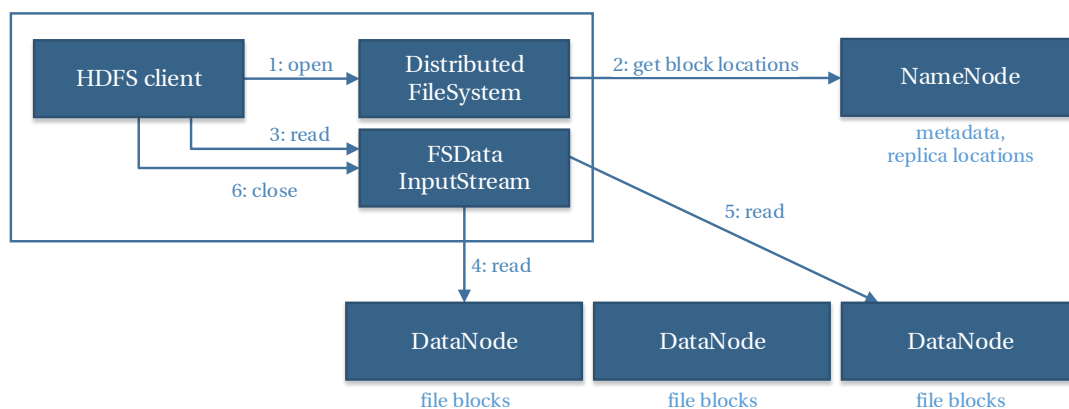


**Figure 2.5:** *Hadoop's architecture [51]*

## HDFS

The Hadoop Distributed File System (HDFS) is an open-source, distributed file system, inspired by the Google File System and written specifically for Hadoop [10]. Unlike other distributed file systems (e.g. Lustre [15]), 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 *DataNodes*.



**Figure 2.6:** *HDFS' architecture*



## **HBase**

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

### **2.3.3 Neo4j**

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

Neo4j is one of the most mature NoSQL databases. It is well documented and provides ample tooling. However, its scalability features are limited: instead of sharding, it only supports replication of data to create a highly available cluster. Of course, the scalability limitations are a hot topic in Neo4j's development. Neo4j's developers make serious efforts to improve Neo4j's scalability in an ongoing project called Rassilon [5].

Neo4j is capable of loading graphs from GraphML [23] and Blueprints GraphSON [14] formats (see Section A.1 for examples). Neo4j graphs can be visualized in Neoclipse, an Eclipse RCP application [16].

## **2.4 Graph Technologies**

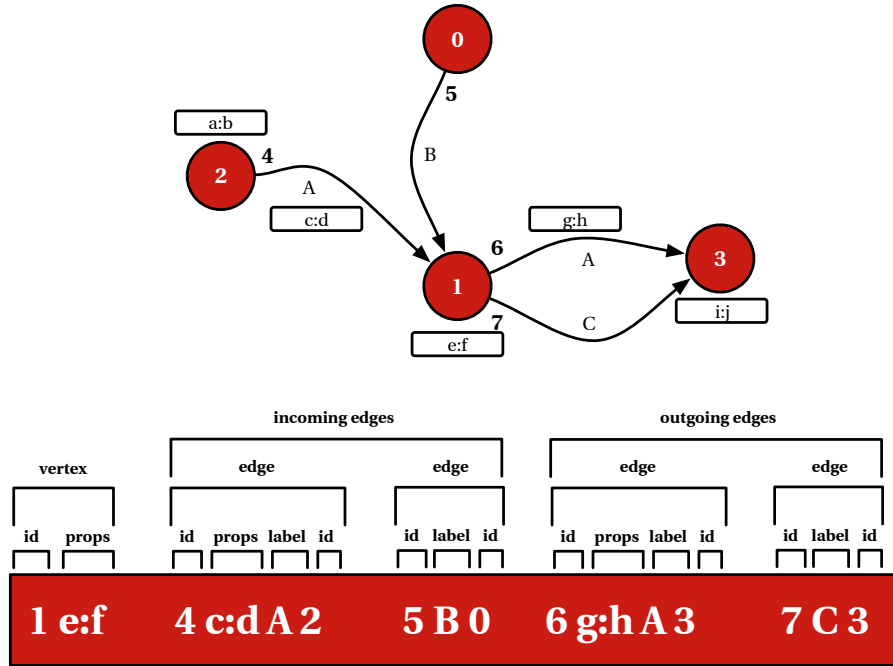
### **2.4.1 Titan**

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

### **Mapping and Sharding**

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

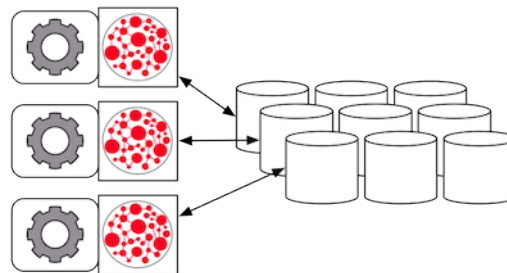
Titan uses the storage backend's partitioner (e.g. Cassandra's RandomPartitioner) to shard the data. A more sophisticated partitioning system that will allow for partitioning based on the graph's static and dynamic properties (its domain and connectivity, respectively) is under implementation as of October 2013, but not yet available.



**Figure 2.7:** Titan graph vertex stored in Cassandra as a row

## Deployment

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



**Figure 2.8:** Using Titan with Cassandra in remote server mode

## Faunus

Although Titan was designed with scalability in mind, its query engine does not work in a parallel fashion. 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.

It is important to note that Faunus always traverses the whole graph and does not use its indices. This makes it slow for retrieving nodes or edges by type (see our typical workload in Section 3.2.2).

### 2.4.2 4store

4store is an open-source, distributed triplestore created by Garlik [6]. Unlike the other tools discussed earlier, 4store is written in C. While 4store is primarily applied for semantic web projects, its maturity and scalability made it an appropriate storage backend for INCQUERY-D's prototype.

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

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

**Table 2.1:** Overview of database technologies

Table 2.1 summarizes the relevant characteristics of the aforementioned database management systems. According 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.

## 2.5 Asynchronous Messaging

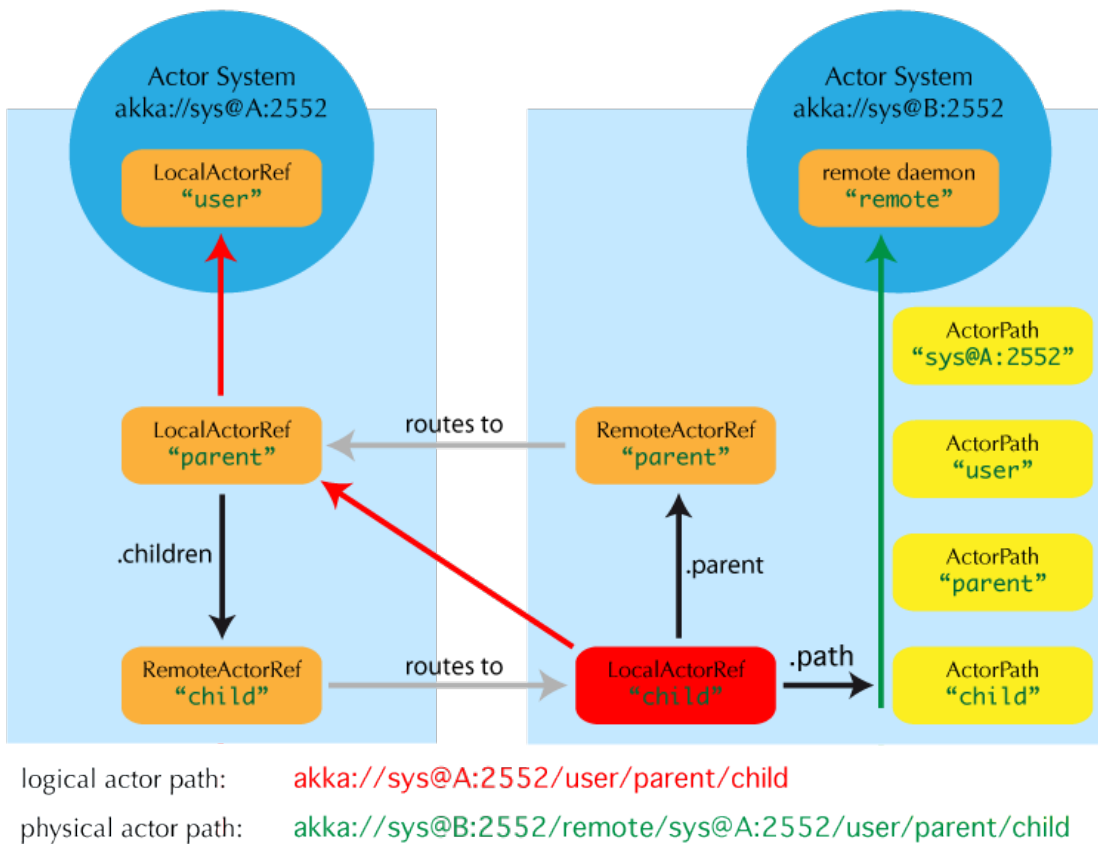
Most distributed, concurrent systems use a messaging framework or message queue service. Because of the nature of the Rete algorithm (Section 3.1.1), INCQUERY-D requires a distributed, asynchronous messaging framework.

### 2.5.1 Akka

Akka is an open-source, fault-tolerant, distributed, asynchronous messaging framework developed by Typesafe [8]. Akka is implemented in Scala, a functional and object-oriented programming language which runs on the Java Virtual Machine. Akka provides language bindings for both Java and Scala.

Akka is based on the actor model [43] 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 2.8). This way, they can be transparently moved between machines on the network.

As of October 2013, the latest version (Akka 2.2) also features *pluggable transport support* to use various transports to communicate with remote systems [8]. For serializing the messages, Akka supports different frameworks, including Java’s built-in serialization, Google Protobuf [21] and Thrift [11].



**Figure 2.9:** Deploying a remote actor in Akka

## 2.6 Eclipse-based Technologies

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

INCQUERY-D’s single workstation predecessor, EMF-INCQUERY is built around Eclipse-based technologies. To reap the benefits of a mutual code base, we designed INCQUERY-D to use as much of EMF-INCQUERY’s components as possible. In the fol-

lowing section, we introduce the Eclipse-based technologies most important for our work.

### 2.6.1 EMF

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

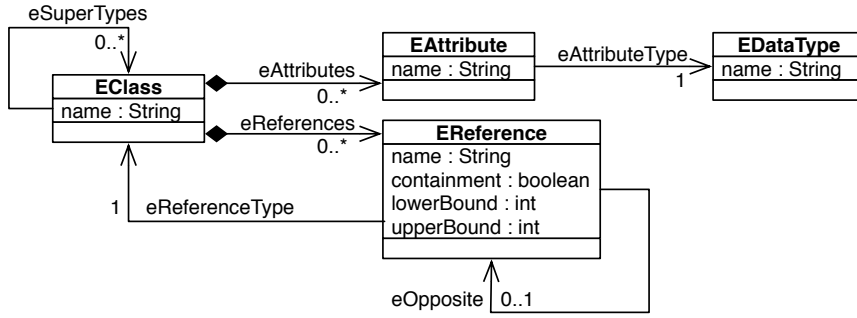
#### Ecore

Ecore is the metamodeling language that is 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 is that it is the *de facto* standard metamodeling environment of the industry, and several domain-specific languages are defined using this formalism.

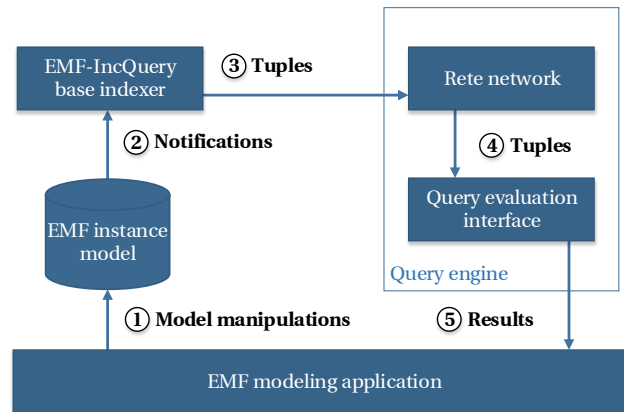
Figure 2.9 illustrates the core elements of the Ecore approach. The full metamodel can be found in the literature. The most important elements are:

- *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.
- *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 is identified by a name. Lower and upper multiplicities can be specified. It is also possible to mark a reference as *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 hierarchisation of the models but the main metamodeling part has been covered here.



**Figure 2.10:** *The Ecore kernel, a simplified subset of the Ecore metamodel*



**Figure 2.11:** *EMF-INCQUERY's architecture*

## 2.6.2 EMF-INCQUERY

The following section is based on [35].

When working with models, it is quite common task to query the model for validating certain properties or searching for interesting parts. EMF-IncQuery [3] provides a solution for this problem: It is a framework to execute fast model queries over EMF models. It is actively developed at Budapest University of Technology and Economics. The current stable version (0.7) proved its usability through industrial use-cases and university researches.

The core of the framework is a query evaluator engine built on top of graph pattern matching engine using Rete algorithm adapted from expert systems to facilitate the efficient storage and retrieval of partial views of graph-like models. In a nutshell, Rete maintains a hierarchical query data structure on top of the model which stores the result of sub-queries. On model change, the event propagates through this data structure leaving the unmodified part of the model untouched. This results in fast, near zero response time and size-independence on small model changes. In return, the model and the query structure has to be loaded into the memory, which can be a significant resource expense.

To access the capabilities of the core, an easy-to-use, type safe API is defined. Using the API, EMF resources and object hierarchies can be loaded and queried incrementally. In addition certain extensions – such as the validation framework – can be attached (Figure 3.2).

Along the API, a complete query language is defined. It provides a declarative way to express the queries over the EMF model 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 framework contains UI tooling which helps the users to effectively develop test and integrate queries into their solution. The first element of the tooling is the rich Xtext [4] based editor for the query language which aids writing well formed queries providing content assist, error markings and such. The next part of the tooling is the ability to load EMF models and execute the queries on them as the user writes them giving visual feedback about the result. The last important part is the code generation. The tooling dynamically generates the source code which contains the programmatic equivalent of the model queries. The users can integrate this code out of the box in Eclipse plugins as well as headless applications to execute queries and get back the results from the source code level.

## 2.7 Challenges

EMF-INCQUERY is proven to be efficient 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  $10^7$  or more elements [57].

Due to the Rete algorithm's memory consumption (Section 3.1.1), EMF-INCQUERY cannot handle large models efficiently [31]. A trivial solution would be to use *vertical scaling*: put 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 [24].

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 4.5). This is a serious problem for MDE, where the queries are typically more complex than in traditional, transactional database management.

Therefore, we decided to implement our own distributed, scalable query layer based on NoSQL and semantic web databases.

### 2.7.1 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 2.2:** *Mapping Ecore to property graphs and RDF*



# 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 Architecture of Incremental Queries: the Single Node Case

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

#### 3.1.1 Incremental Query Evaluation

Some queries, e.g. well-formedness constraints in MDE are evaluated many times, while the data sets 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 occurred.

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.

### Incremental Pattern Matching Algorithms

Numerous algorithms were implemented 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 in the graph<sup>1</sup>. TREAT [52] 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 [28] 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.

### Rete in General

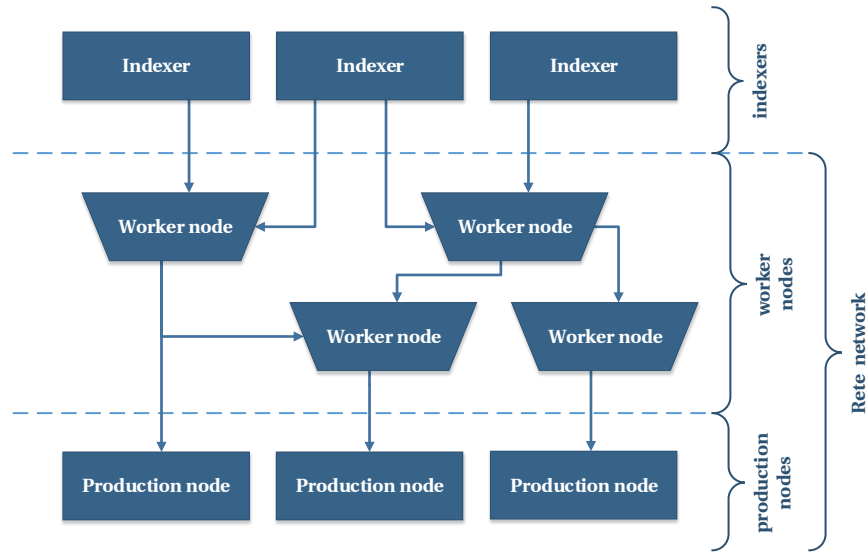
The algorithm was originally created by Charles Forgy [37] for rule-based expert systems. Gábor Bergmann adapted the algorithm for EMF models and added many tweaks and improvements to it [29].

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 ( $\pi$ ), selection ( $\sigma$ ), join ( $\bowtie$ ) and antijoin ( $\bowtie^c$ ) 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 net is built on top of type-specific indexers, which are responsible for providing quick lookups and generating notifications for the worker nodes.

---

<sup>1</sup> *Rete* is Latin for *net* or *comb*.



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

### 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).

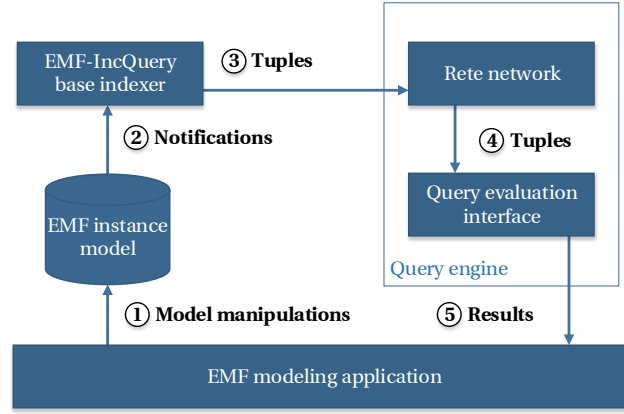
### Notification Mechanisms

*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.

#### 3.1.2 Rete in EMF-INCQUERY

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

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.



**Figure 3.2:** EMF-INCQUERY's *architecture*

### Query optimization

Similarly to optimizing query plans for relational databases, we can also optimize the Rete net'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 Rete node. More details are available in [30].

### Data representation

The Rete network represents the data in *tuples*. Basically, in the Rete network used in EMF-INCQUERY, the tuples can contain two sorts of values:

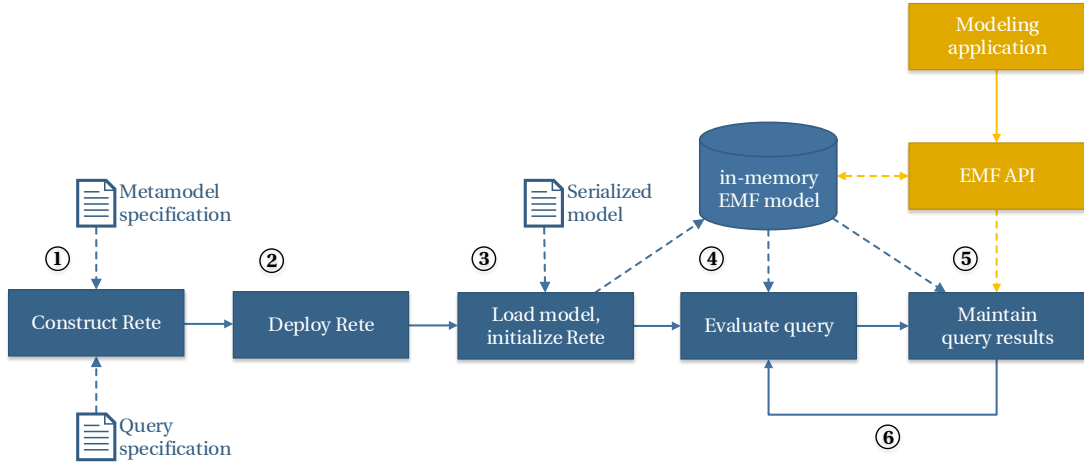
- pointers to an EMF model,
- Ecore scalar values (EString, EInt, etc. instances).

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 ( $\pi$ ) and join ( $\bowtie$ ), can be simply defined using tuple masks [29].

### Workflow

In the following, we describe EMF-INCQUERY's workflow based on Figure 3.3.

Based on the *metamodel* and the *query specification*, INCQUERY-D 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 indexes. 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 net, hence keeping it in a consistent state ⑤. The query results

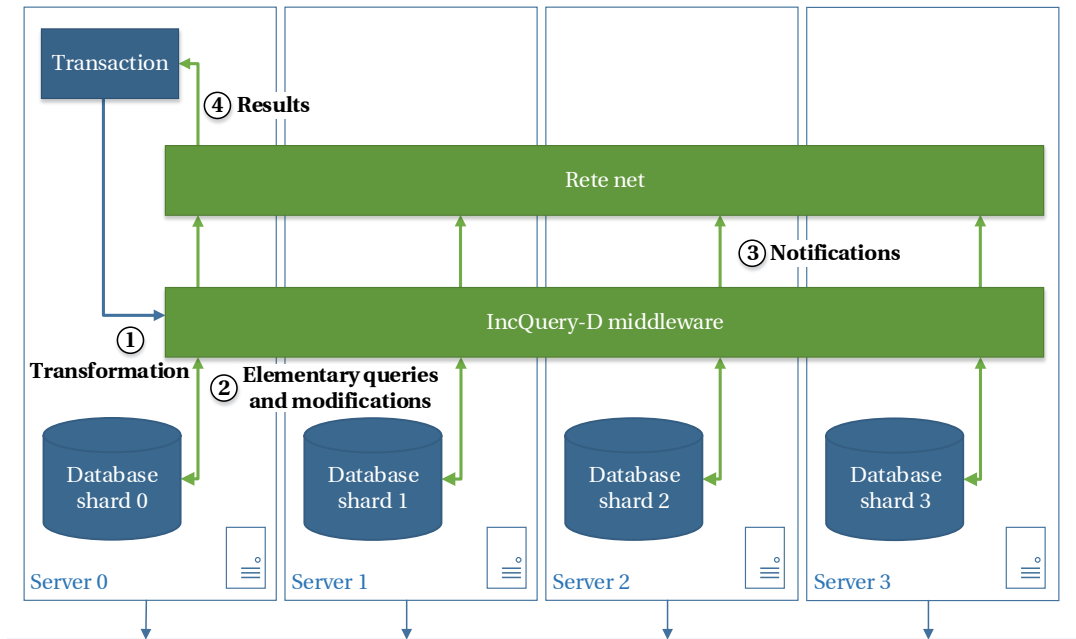


**Figure 3.3:** EMF-INCQUERY's *workflow*

can be retrieved from the Rete net ⑥. The modeling application may modify the model and reevaluate the query again.

### 3.2 Extensions for Distributed Scalability

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

### 3.2.1 INCQUERY-D's Architecture

The INCQUERY-D architecture in an example configuration is shown in Figure 3.4. 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 3.2.3). 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 3.1.1) to the production network and retrieves the query results from the production network ④. The *production network* is implemented with a distributed Rete net which provides incremental query evaluation (Section 3.1.1).

### 3.2.2 Indexing and Initialization

Indexing is a common technique for decreasing the execution time of database queries. In MDE, *model indexing* is the key to high performance model queries. As MDE primarily uses a metamodeling infrastructure, all queries utilize some type attribute. In property graphs, the equivalents are node type properties and edge labels (see also Section 3.3.1). Typical elementary model queries are the following:

- Retrieving all node instances of a given type (e.g. get all nodes with the type Person).
- Retrieving all edges 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).

To process these queries 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 3.4 ②). In order to reduce the initialization time, the underlying storage layer must be able to process these queries efficiently.

### 3.2.3 Distributed Storage Layer

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 possi-

ble. 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.

### 3.2.4 Notification in Distributed Database Management Systems

While relational databases usually provide *triggers* 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 current implementation, notifications are controlled by the middleware by providing a facade for all model manipulation operations (Figure 3.4 ③). The notification messages are propagated through the Rete network via the Akka messaging framework.

### 3.2.5 Graph-like Data Manipulation

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

### Data Representation

Conceptually, the architecture of INCQUERY-D allows the usage of a wide scale of model representation formats. Our prototype has been evaluated in the context of the *property graph* and the *RDF* data model, but other mainstream metamodeling and knowledge representation languages such as relational databases' SQL dumps and Ecore instance models Section 2.6.1 could be supported, as long as they can be mapped to an efficient and distributed storage backend (e.g. triplestores, key-value stores or column-family databases).

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

### 3.2.6 Degrees of Freedom

The Rete algorithm (Section 3.1.1) 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 net. 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 chosen 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.

### 3.2.7 Distributed Termination Protocol

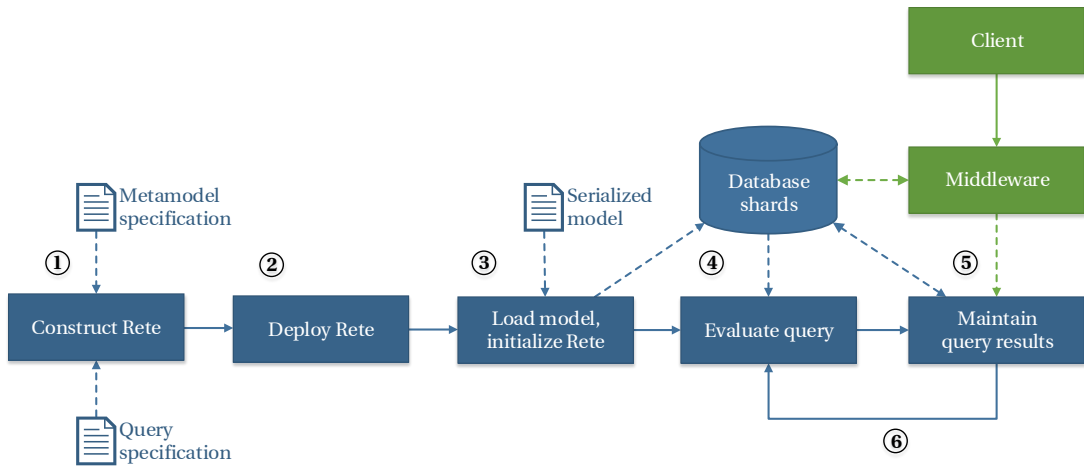
INCQUERY-D's current termination protocol works by adding a stack to the message. The stack registers each Rete node the message passes through. After the message reaches the production node, the termination protocol starts. Based on the content of the stack, acknowledgement messages are propagated back on the network. When all



relevant indexer nodes (where the original notification token(s) started from) receive the acknowledge messages, the termination protocol finishes.

### 3.2.8 Workflow

In the following part, we will describe the workflow behind the pattern matching process. Starting from a metamodel, an instance model and a graph pattern, we will cover the problem pieces that need to be solved for setting up an incremental, distributed pattern matcher. The workflow is shown on Figure 3.5.



**Figure 3.5:** *The workflow of INCQUERY-D*

By design, INCQUERY-D’s workflow’s steps are similar to EMF-INCQUERY’s, discussed in Section 3.1.2. However, due to the system’s distributed nature, they are more difficult to design and implement. In INCQUERY-D, deploying the Rete network ② requires the deployment of remote actors Section 2.5.1 on the servers. Both the Rete indexers and the database are distributed across the cluster. Hence, loading the model and initializing the Rete net needs network communication ③. The Rete net works using Akka’s remote messaging feature. The query results can be retrieved from the Rete net (this may also require network communication) ④. The database shards can only be accessed through the middleware, which is responsible for sending notifications to the Rete net’s appropriate indexers. After the notifications are processed and the termination algorithm finishes, the Rete net is in a consistent state ⑤. The results can be retrieved by the client and it may modify the model and reevaluate the query again ⑥.

### 3.3 Tooling

To be able to focus on the distributed aspects, of the system, we aimed to build INCQUERY-D on top of EMF-INCQUERY’s pattern language (IQPL) and its Rete net gen-

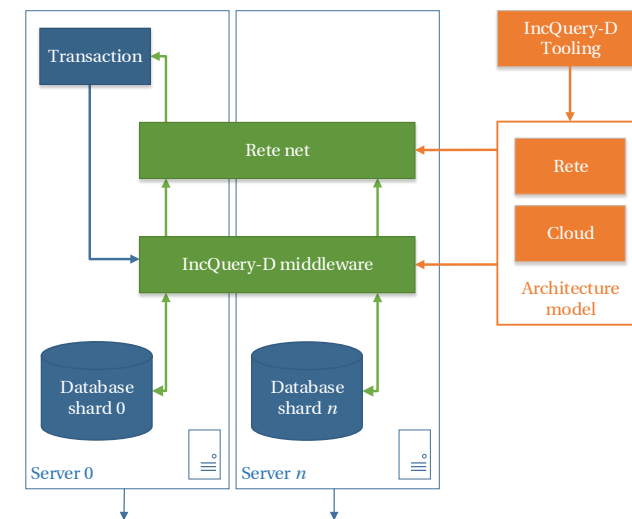
erator. Also, EMF-INCQUERY has an Eclipse-based user interface for defining and executing queries.

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.

### 3.3.1 Runtime Model-based Dashboard

To aid the system's dynamic capabilities, we plan to implement 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 net and initializing the middleware (Figure 3.6).



**Figure 3.6:** Architecture of INCQUERY-D with a runtime dashboard

To provide live feedback, we plan to implement 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.

## 3.4 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.

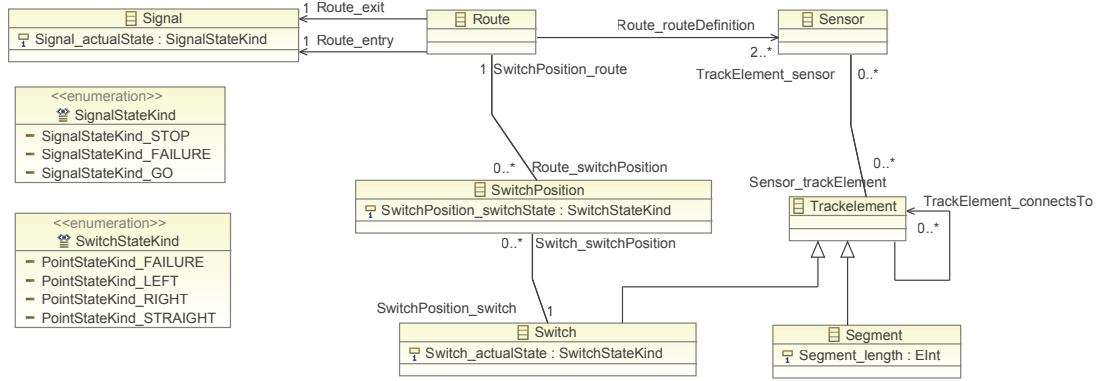


Figure 3.7: The EMF metamodel of the railroad system

### 3.4.1 Case Study: Railroad System Design

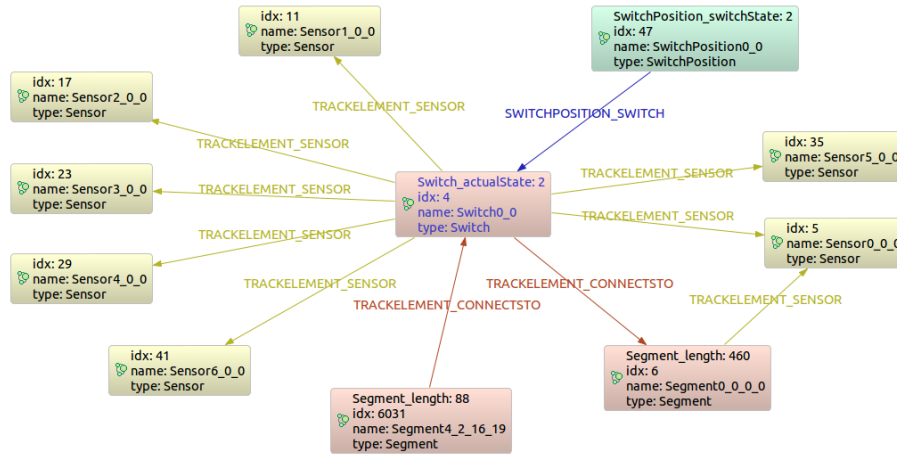


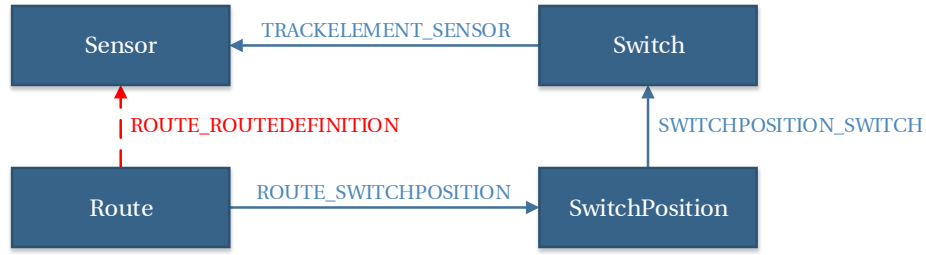
Figure 3.8: A subgraph of a railroad system visualized

The example is built around an imaginary railroad system. The system's network is composed of typical railroad items, including signals, segments, switches and sensors. The complete EMF metamodel is shown on Figure 3.7. A subgraph of an instance model is shown on Figure 3.8.

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 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

The *RouteSensor* query looks for Sensors 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 3.9. Basically, the RouteSensor



**Figure 3.9:** Graphical representation of the RouteSensor query's pattern. The dashed red arrow defines a negative application condition.

query binds the type of the vertices, defines three edges and one negative edge, called NAC (negative application condition).

```

1 package hu.bme.mit.train.constraintcheck.incquery
2
3 import "http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl"
4
5 pattern routeSensor(Sen, Sw, Sp, R) = {
6   Route(R);
7   SwitchPosition(Sp);
8   Switch(Sw);
9   Sensor(Sen);
10
11   Route.Route_switchPosition(R, Sp);
12   SwitchPosition.SwitchPosition_switch(Sp, Sw);
13   Trackelement.TrackElement_sensor(Sw, Sen);
14
15   neg find head(Sen, R);
16 }
17
18 pattern head(Sen, R) = {
19   Route.Route_routeDefinition(R, Sen);
20 }

```

**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<sup>2</sup>.

```

1 PREFIX base: <http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#>
2 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3 PREFIX owl: <http://www.w3.org/2002/07/owl#>

```

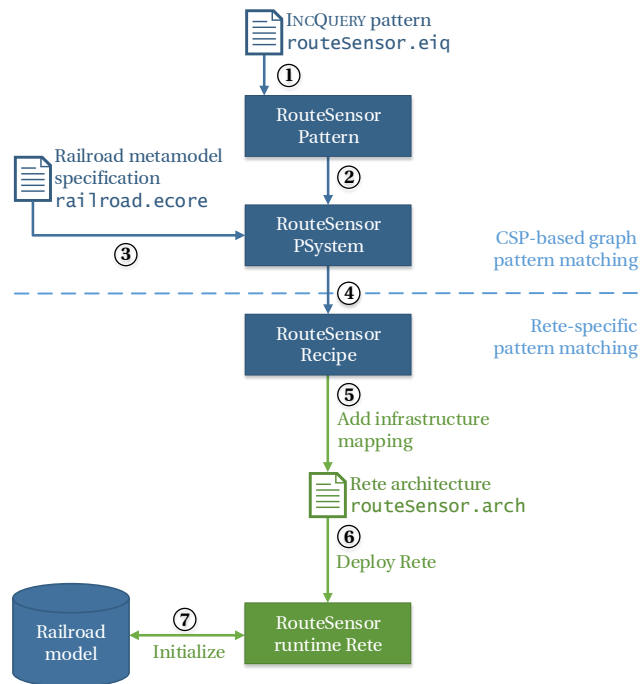
<sup>2</sup>Note that the two queries are slightly different: the SPARQL query returns only a set of Sensors, while the IQPL query returns a set of (Sensor, Switch, SwitchPosition, Route) tuples.

```

4 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
5
6 SELECT DISTINCT ?xSensor
7 WHERE
8 {
9     ?xRoute rdf:type base:Route .
10    ?xSwitchPosition rdf:type base:SwitchPosition .
11    ?xSwitch rdf:type base:Switch .
12    ?xSensor rdf:type base:Sensor .
13    ?xRoute base:Route_switchPosition ?xSwitchPosition .
14    ?xSwitchPosition base:SwitchPosition_switch ?xSwitch .
15    ?xSwitch base:TrackElement_sensor ?xSensor .
16
17    FILTER NOT EXISTS {
18        ?xRoute ?Route_routeDefinition ?xSensor .
19    } .
20 }

```

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



**Figure 3.10:** *INCQUERY-D's workflow*

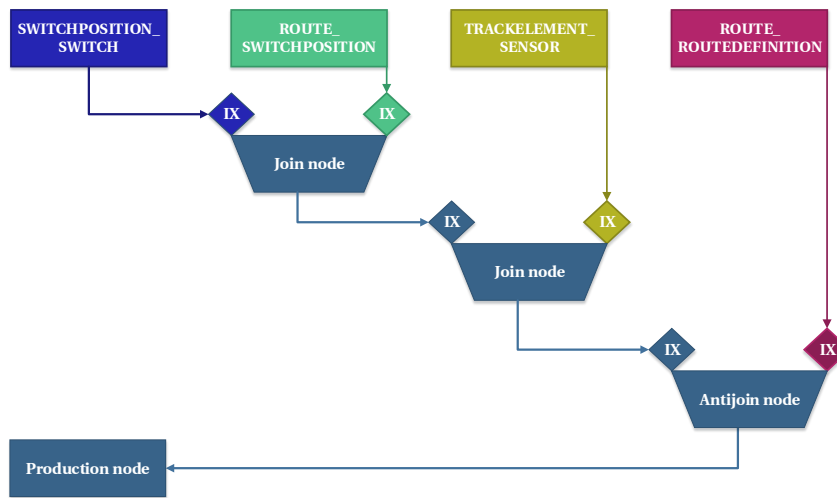
### 3.4.2 Workflow of the Example

Following the workflow defined in Section 3.2.8, we will cover the steps for deploying and operating a distributed pattern matcher for the *RouteSensor* query.

## Constructing a Rete net

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 3.7. Based on the query ② and the metamodel ③ EMF-INCQUERY builds a *pattern system* (PSys-tem). The PSys-tem 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 3.11.



**Figure 3.11:** The RouteSensor query's layout

## Deploying the Rete Net

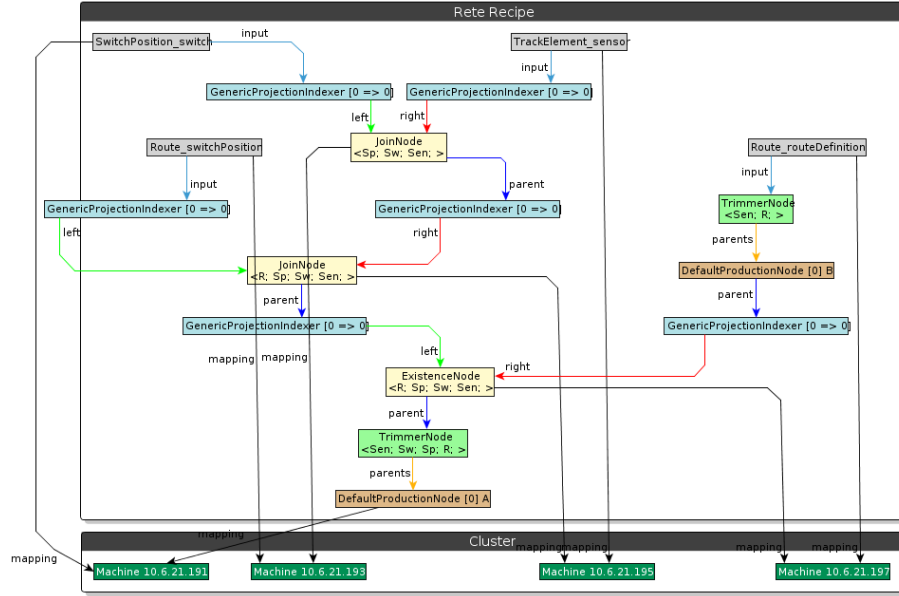
The Rete nodes are allocated to the cluster's servers by providing the infrastructure mapping ⑤.

In INCQUERY-D's current implementation, the Rete recipe's nodes are allocated manually on the cloud servers (called *Machines*). 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 3.12). The Rete network is deployed to the Akka instances running on the servers ⑥.

## Evaluating Query

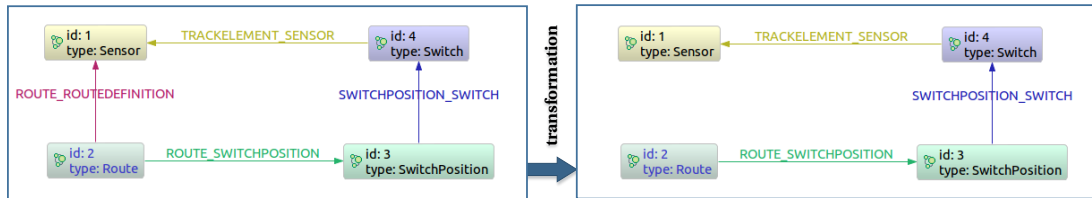
The query is evaluated by initializing the Rete net ⑦ and reading the results from its production node.



**Figure 3.12:** The yFiles viewer in INCQUERY-D's tooling

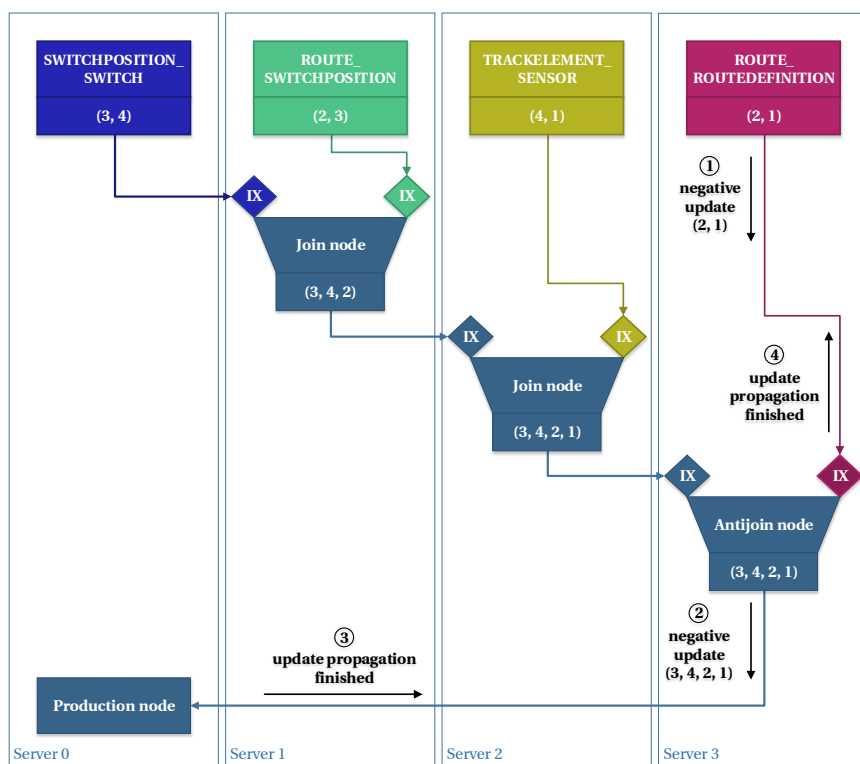
### Maintaining the Query Results

In order to provide query results that are consistent with the model, we need maintain the Rete net's state. Suppose we have the graph shown on the left side of Figure 3.13 loaded to the Rete net and we decide to delete the ROUTE\_ROUTEDEFINITION edge between vertices 2 and 1.



**Figure 3.13:** A modification on a TrainBenchmark instance model

Figure 3.14 shows the distributed Rete net 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 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 tuples that have been added or deleted since the last modification.



**Figure 3.14:** Operation sequence on a distributed Rete net



# Chapter 4

## Evaluation of Scalability and Performance



We ~~implemented~~ a prototype of INCQUERY-D to evaluate the feasibility of the approach. In the following chapter, we introduce the TrainBenchmark, a performance benchmark for query engines and the distributed TrainBenchmark, which is also capable of measuring certain scalability aspects. We present the benchmark environment and analyze the results.



### 4.1 Goals

We ~~implemented~~ a prototype of INCQUERY-D 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.



#### 4.1.1 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 goal of our benchmark was to prove that an INCQUERY-D cluster is indeed capable of processing queries for large models.



## 4.2 ~~TrainBenchmark~~



The TrainBenchmark was designed by Benedek Izsó, Zoltán Szatmári and István Ráth [48] to measure the efficiency of model queries and manipulation operations in different tools. The TrainBenchmark is primarily targeted for typical MDE workloads, more specifically for well-formedness validations.

The benchmark is built on the *railway system metamodel*, defined in Section 3.4.



### 4.2.1 Benchmark Goals

The TrainBenchmark measures both the response time and the scalability of the tools. 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 instant feedback after each edit. Therefore, we would like to run re-evaluate well-formedness queries quickly, ~~which implies that incremental query engines are more favorable for this workload.~~



The benchmark defines four distinct phases:

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

To measure 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 4.2.3). Running the same validation sequence on different model sizes highlighted the limitations of the tested query engines.

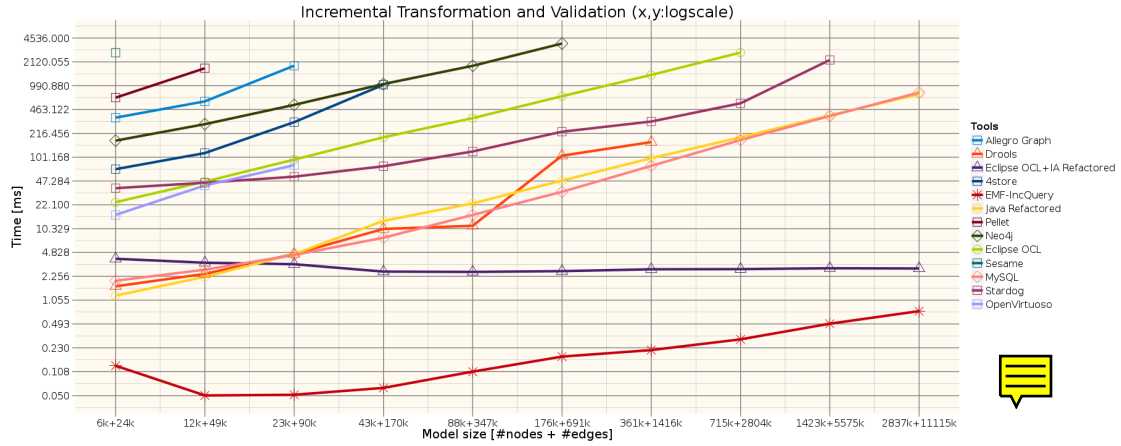
Scalability is also measured along the complexity of the queries. The benchmark defines four queries, each testing different aspects of the query engine (filtering, join and antijoin operations, etc.).

### 4.2.2 Results



The TrainBenchmark was ~~implemented for~~ different tools originating from various technological spaces, e.g. EMF-based tools (EMF-INCQUERY, Eclipse OCL), semantic web technologies (Allegro Graph, Sesame, ~~4store~~), NoSQL databases (Neo4j), etc.





**Figure 4.1:** TrainBenchmark results measured on a single node

Figure 4.1 shows the incremental transformation and validation time for the *Route-Sensor* query, discussed in Section 3.4.1. The results clearly show the advantage of incremental query engines. Both Eclipse OCL Impact Analyzer and EMF-INCQUERY scale sublinearly (moreover, their characteristic is almost constant), while non-incremental tools scale linearly at best, which renders them inefficient for large models.

### 4.2.3 Generating Models

Due to both confidentiality and technical reasons, it is difficult to obtain real-world industrial models and queries. Also, using confidential data sets hamstrings the reproducibility of the conducted benchmarks. Therefore, we generated instance models which mimic real-world models.

The instance models are generated pseudorandomly, with pre-defined structural constraints [46]. The generator is capable of generating models of different sizes and formats, including EMF, OWL, RDF and SQL.

As mentioned earlier, the railroad system metamodel, introduced in Section 3.4.1, was designed in Ecore. Based on the mapping from the Ecore kernel (Section 3.3.1) we created the equivalent instance models for property graphs as well.

The Ecore kernel concepts map to property graphs as follows:

- 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, are defined with the Java primitive type `int`.


Note that the property graph and the RDF data models lack some of the advanced metamodeling features of Ecore. For example, an Ecore `EReference` not just defines a relationship between specific `EClass` instances (this is already impossible in a property graph), but also defines multiplicity and containment constraints. This cannot be enforced by the property graphs and RDF data models, instead, it has to be handled by the modeling application explicitly.

### 4.3 Distributed **TrainBenchmark**


Based on the `TrainBenchmark`, discussed in Section 4.2, we created an extended version for distributed systems. The main goal of the distributed `TrainBenchmark` is the same as the original's: measure the response time and scalability of different tools.

#### 4.3.1 Generating 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. 

#### 4.3.2 Distributed Architecture

The distributed benchmark defines the same phases as the original `TrainBenchmark`. 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 also run distributedly.** 

#### 4.3.3 Benchmark Limitations

A common reason for designing and implementing distributed systems is that they are capable of handling a great number of concurrent requests. This way, more users can

use the system at the same time. In the distributed TrainBenchmark, the system is only used by a single user. Simulating multiple users and issuing concurrent requests is subject to future work (Section 6.3).

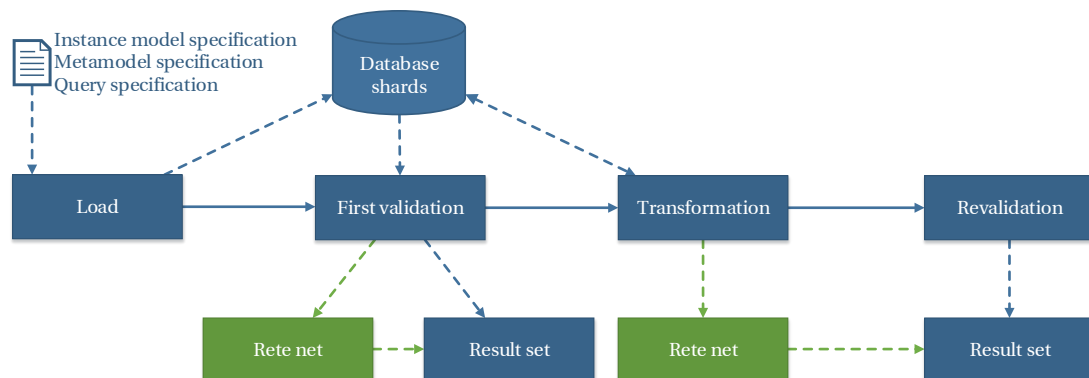
## 4.4 Benchmark Environment

We used the distributed TrainBenchmark (Section 4.3) to evaluate INCQUERY-D’s performance and compare it to non-incremental solutions. Our benchmark environment was very similar to the one used for evaluating INCQUERY-D’s earlier prototype [45].

In the following section, we will discuss the benchmark setup and the environment in detail.

#### 4.4.1 Benchmark Setup

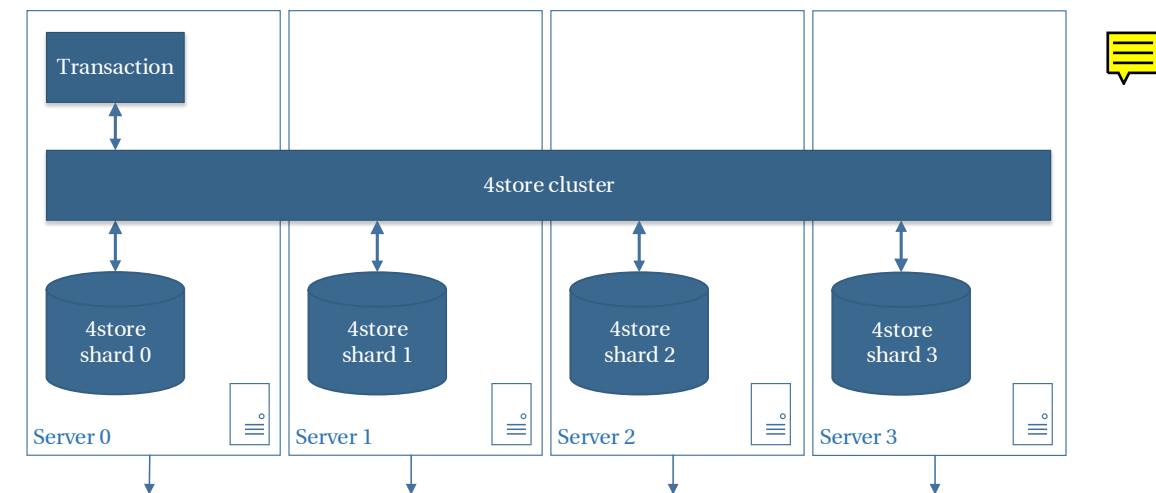
We tested INCQUERY-D with two storage backends: 4store (Section 2.4.2) and Titan (Section 2.4.1). As a *non-incremental baseline*, we used 4store as a standalone database management system. While we also planned to use Titan, we found that Titan is more geared towards local graph traversals. Although graph patterns can be formulated as traversal operations, we found that even for small graphs, the system was unable to run the queries efficiently.



**Figure 4.2:** *The benchmark scenario. The green arrows and components are specific to the incremental scenario.*

The benchmark follows the phases defined in the distributed TrainBenchmark. The workflow is shown on Figure 4.2. Note that the main difference between the batch and incremental scenarios is that the latter maintain a distributed Rete net, which allows efficient query (re)evaluation.

The architecture of INCQUERY-D is discussed in detail in Section 3.2. The baseline scenario’s benchmark setup is shown on Figure 4.3.



**Figure 4.3:** *The non-incremental "batch" scenario benchmark's setup*

#### 4.4.2 Hardware and Software Ecosystem

As the testbed, we deployed our system to a private cloud consisting of 4 virtual machines on separate host computers. The private 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. Our countermeasures are discussed in Section 4.6.2.

The detailed configuration of the servers are provided below.

##### Hardware

Each virtual machine used two cores of an Intel Xeon L5420 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.

- Operating system and Java Virtual Machine
  - Ubuntu 12.10 64-bit
  - Oracle Java 7 64-bit
- Database management system
  - Neo4j 1.8

- 4store 1.1.5
  - Titan 0.3.2
    - \* Faunus 0.3.2
    - \* Hadoop 1.1.2
    - \* Cassandra 1.2.2
- Messaging framework
  - Akka 2.1.2
- Development environment and tooling
  - Eclipse 4.3 (Kepler)

#### 4.4.3 Benchmark Methodology and Data Processing

The benchmark coordinator software used the TrainBenchmark’s framework to collect the data about the results of the benchmark. We ~~mainly~~ 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 collected in text files, which were processed by an R script [25] ~~developed by Benedek Izsó~~. The script is capable of generating and visualizing the results.

We also worked to ensure the *functional equivalence* of the measured tools. During the development, we followed the TrainBenchmark’s well-defined specification [46]. In runtime, we checked the result set provided by each tool.

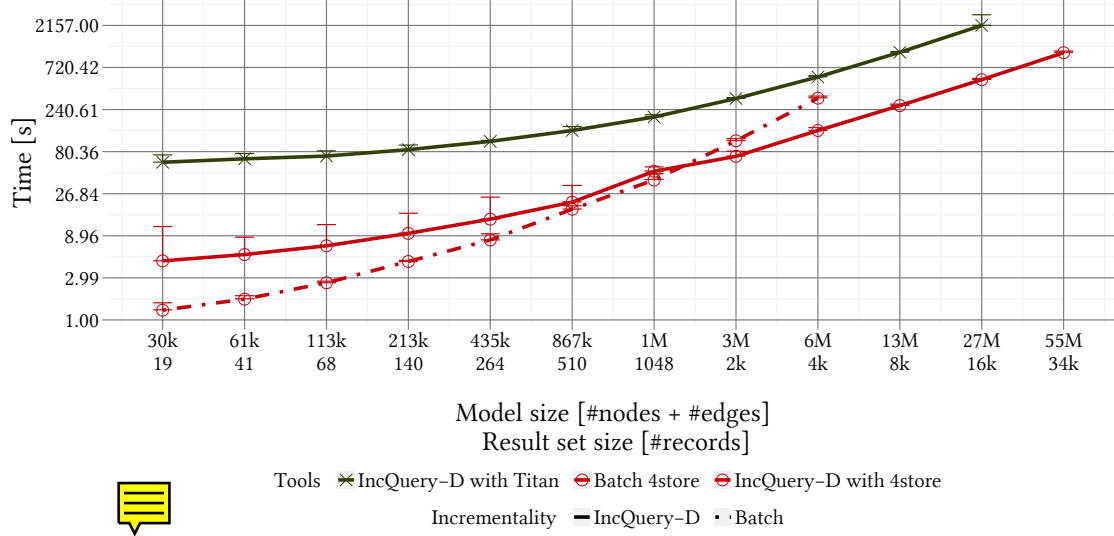
### 4.5 Results

In the following section, we discuss the results of our benchmark.

#### 4.5.1 Result visualizations

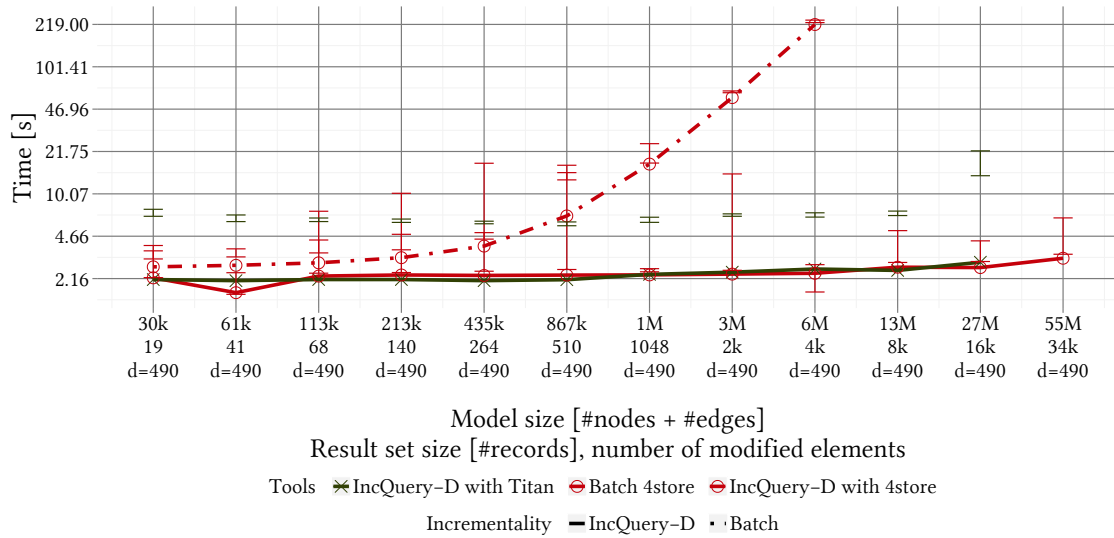
We present the visualizations of the benchmark’s most important results. The  $x$  axis shows the models size (number of model elements), while the  $y$  axis shows execution time. Both axes are logarithmic. The *batch* (non-incremental) tool is represented with a dashed line, while the *incremental tools* are represented with solid lines.

The execution times for the *load and first validation* phases are shown on Figure 4.4. As expected, due to the overhead of the Rete network’s construction, the *batch* tool is



**Figure 4.4:** Execution times for load and first validation

faster for small models. However, it is important to observe that even for medium-sized models (with a couple of million elements), the *incremental* tools start to edge ahead. This shows that the Rete network's construction overhead already pays off for the first validation.

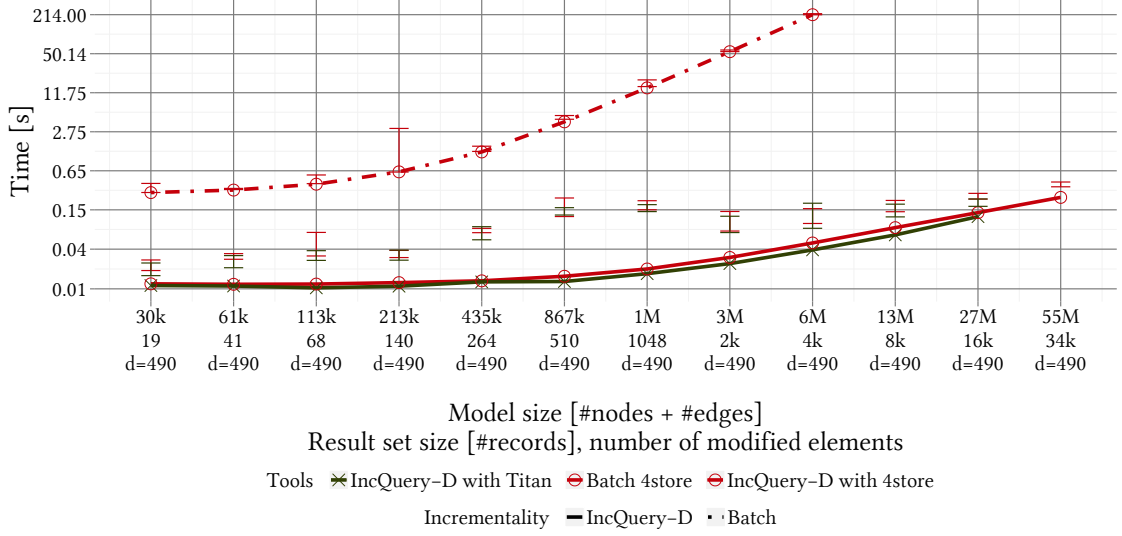


**Figure 4.5:** Execution times for transformation

The execution times for the *transformation* phase are shown on Figure 4.5. The incremental tools provide faster query transformation times due to the fact that instead of querying the database, the modeling application can rely on the query layer's indexes. Even for medium-sized models, the incremental tool are more than two orders of magnitude faster than the batch tool.

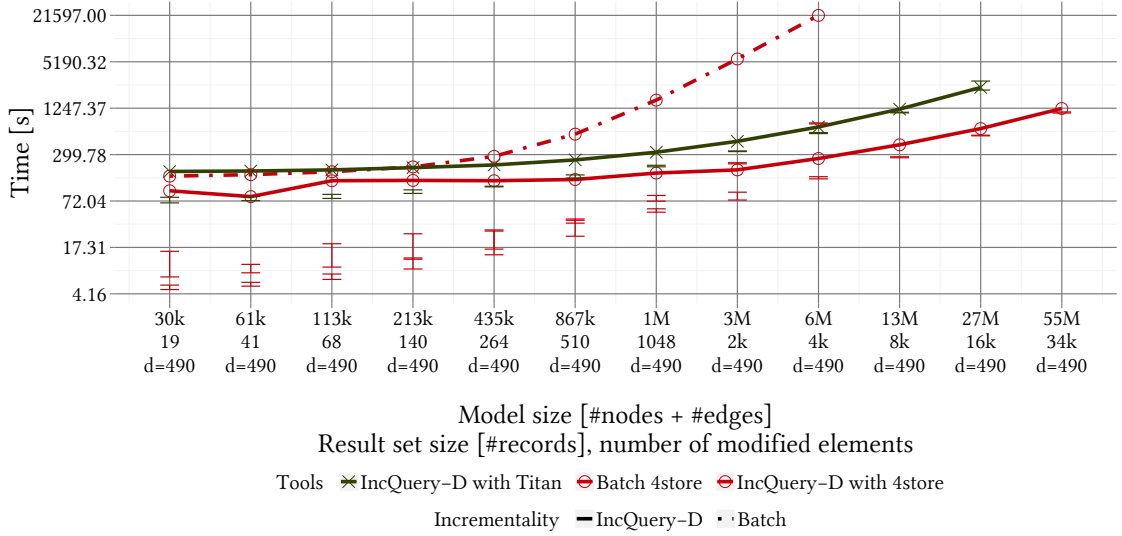
The incremental tools have an even greater advantage for *revalidation* times, shown





**Figure 4.6:** Execution times of the revalidation

on Figure 4.6. For medium-sized models, they are more than three orders of magnitude faster than the batch tool. **Note that the incremental tools provide sub-second revalidation times even for large models with tens of millions of elements.**



**Figure 4.7:** Total execution times for 50 validations

Figure 4.7 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, 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.

The results show that the 4store-based INCQUERY-D implementation is consistently faster in the *load* phase than the Titan-based one. This is due to 4store's simpler archi-

texture and different data model, which is better suited to the INCQUERY-D middle-ware’s elementary model queries.

## 4.6 Result Analysis

The results clearly show that the initialization of the Rete net 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 net.

The almost constant characteristic of execution times of the incremental tools’ *transformation* and *validation* phases confirm that a distributed, scalable, incremental pattern matcher is feasible with current technologies. The results also show 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 4.1).



Another important observation is that for incremental tools, the execution time is approximately proportional to *the number of affected model elements*. 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.

### 4.6.1 Memory Consumption

During our experiments, we ran into cases where the Java Virtual Machine ran out of or had just enough memory, resulting in `OutOfMemoryError: Java heap space` and `OutOfMemoryError: GC overhead limit exceeded` exceptions, respectively. Introducing a fault-tolerance mechanism for these cases is subject to future work (Section 6.3).

### 4.6.2 Threats to Validity

To guarantee the correctness of our benchmarks, we added 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. 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 days or months).



As discussed in Section 4.4.2, our servers could be influenced from the workload caused by other users of the same cloud. To minimize 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.

## 4.7 Benchmark Results with Neo4j



During the earlier phases of the research, we conducted measurement using only Neo4j [47]. The benchmark's setup was slightly different, with the main differences being the following.

- Due to the lack of sharding in Neo4j, we *sharded the graph manually*. This had a number of 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 net. The elementary model queries (for filling the indexers) were ran on all shards separately and aggregated by the indexers. The transformations also ran separately.
- The servers had twice as much memory (16 GB).



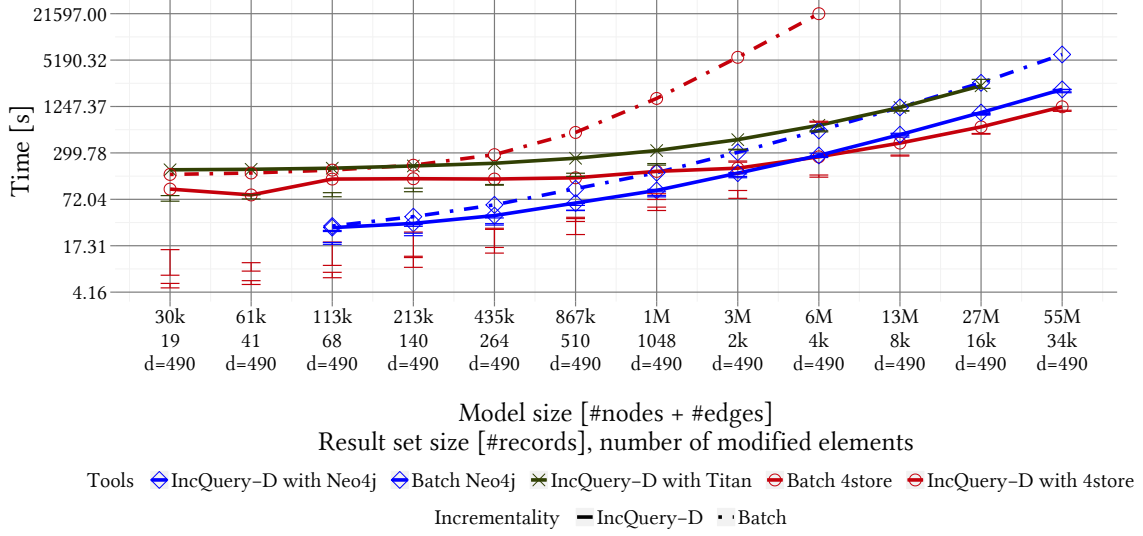
Despite the differences, we can still derive some useful conclusions by compare the results of this benchmark with the most recent one detailed previously. Figure 4.8 shows that the Neo4j-based incremental tool consistently outperforms the Neo4j's own query engine.

Figure 4.9 shows that while Neo4j's non-incremental query engine scales better than 4store's (in this particular setup), the incremental tool based on Neo4j is about two orders of magnitude faster.

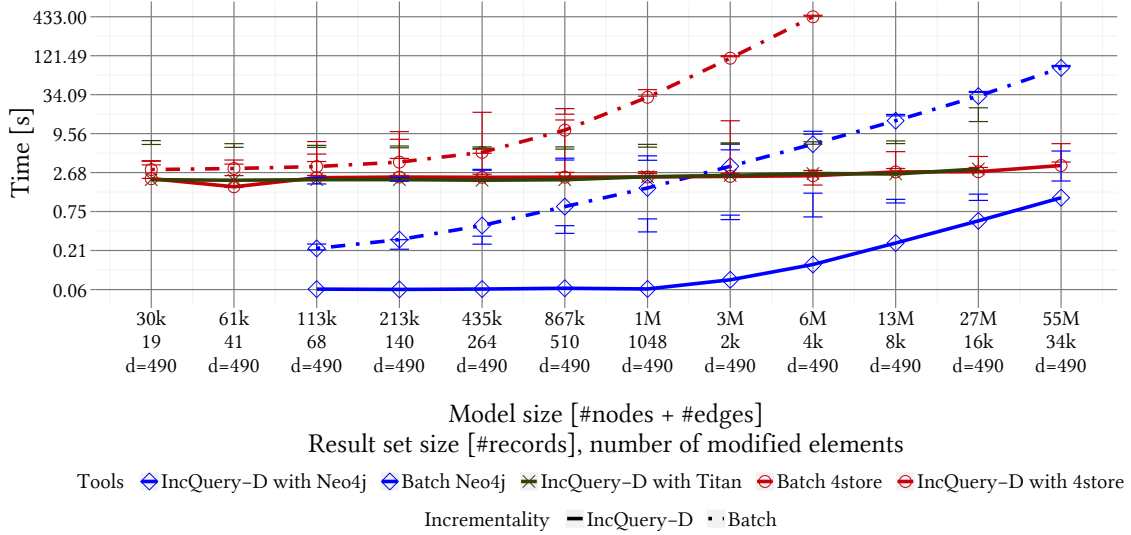
## 4.8 Summary



Our benchmarks proved that the concept of a distributed, incremental pattern matcher is a working one. INCQUERY-D's scalability characteristics confirmed that while network overhead is present, 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.



**Figure 4.8:** Total execution times for 50 validations



**Figure 4.9:** Execution times for transformation and revalidation

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 ??), 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 [36] engine. To address scalability issues, *incremental* impact analysis tools [39] 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 [55] uses this algorithm to perform complex event processing (formulated in SPARQL) on RDF data, gathered from distributed sensors.

Diamond [53] 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 [31], 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 [27]. 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.

# Chapter 6

## Conclusions

This chapter, ...

### 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, and its feasibility has been evaluated using a benchmarking scenario of on-the-fly well-formedness validation of software design models. The results are promising as they show nearly instantaneous query re-evaluation as model sizes grow well beyond  $10^7$  elements.

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

#### 6.1.1 Scientific Contributions

- I implemented a *distributed, asynchronous version of the Rete algorithm* with a termination protocol. Based on the Rete algorithm, I created a *distributed incremental query engine's prototype*, which is not only detached from the data storage backend, but also agnostic to the storage backend's data model. To prove this, the query engine was tested with both property graphs and RDF graphs.
- Based on EMF-INCQUERY and István Ráth's work, I created an *Eclipse-based environment* to provide automated deployment of the Rete network. This allows the user to define complex queries in IQPL, a high-level pattern language.
- I conducted a benchmark to measure INCQUERY-D's *response time and scalability characteristics*. For the benchmark's baseline, I created *distributed non-incremental benchmark scenarios*, with Neo4j and 4store.

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

### 6.1.2 Practical Accomplishments

- I extended the TrainBenchmark with a *new instance model generator*, which can produce property graphs and serialize them in various formats: GraphML, Blueprints GraphSON and Faunus GraphSON.
- I extended the TrainBenchmark to work in a distributed environment.
- I implemented INCQUERY-D's prototype, including the storage, the middleware and the query layer. I wrote more than 3000 lines of Java code and approximately 500 lines of configuration and deployment scripts.
- I experimented with modern non-relational database management systems with a focus on NoSQL graph databases and triple stores. I created the connector class in INCQUERY-D's middleware and formulated the necessary the Gremlin queries.
  - 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 implemented scripts to install the *4store triplestore* on a cluster. I created the connector class in INCQUERY-D's middleware and formulated the necessary the SPARQL queries.
  - I deployed a manually sharded *Neo4j cluster*. I created the connector class in INCQUERY-D's middleware to access Neo4j and formulated the appropriate Cypher queries.
- I implemented scripts for *automating the benchmark* and *operating a cluster of Akka microkernels*.

## 6.2 Limitations

- The Rete nodes are manual allocated.
- Only a subset of the Rete nodes are implemented. No *check* expressions.
- Tooling does not cover the whole workflow.



### 6.3 Future work

For future work, we plan on providing more sophisticated automation for sharded Ecore models, and further exploring advanced optimization challenges such as dynamic reconfiguration and fault tolerance.

We also plan experiment with programming languages that are better suited to asynchronous algorithms (e.g. Erlang and Scala) and try different database systems (e.g. MongoDB) as our storage layer.

Distributed TrainBenchmark w/ concurrent requests.

- Use programming languages better suited for asynchronous processing, e.g. Scala, a Java-based functional object-oriented programming language.
- Implement *node sharding*.
- Automatic allocation of Rete nodes using CSP (Constraint Satisfaction Problem), DSE (Design Space Exploration) [42]
- Implement *live monitoring*.

# List of Figures

2.1	Different graph data models (based on [56]) . . . . .	11
2.2	The TinkerPop software stack [12] . . . . .	13
2.3	Cassandra's ring for data partitioning [7] . . . . .	15
2.4	Hadoop's architecture [51] . . . . .	16
2.5	HDFS' architecture . . . . .	16
2.6	Titan graph vertex stored in Cassandra as a row . . . . .	18
2.7	Using Titan with Cassandra in remote server mode . . . . .	18
2.8	Deploying a remote actor in Akka . . . . .	20
2.9	The Ecore kernel, a simplified subset of the Ecore metamodel . . . . .	22
2.10	EMF-INCQUERY's architecture . . . . .	22
3.1	The structure of the Rete propagation network . . . . .	26
3.2	EMF-INCQUERY's architecture . . . . .	27
3.3	EMF-INCQUERY's workflow . . . . .	28
3.4	INCQUERY-D's architecture on a four-node cluster . . . . .	28
3.5	The workflow of INCQUERY-D . . . . .	32
3.6	Architecture of INCQUERY-D with a runtime dashboard . . . . .	34
3.7	The EMF metamodel of the railroad system . . . . .	34
3.8	A subgraph of a railroad system visualized . . . . .	35
3.9	Graphical representation of the RouteSensor query's pattern. The dashed red arrow defines a negative application condition. . . . .	35
3.10	INCQUERY-D's workflow . . . . .	37
3.11	The RouteSensor query's layout . . . . .	38
3.12	The yFiles viewer in INCQUERY-D's tooling . . . . .	39
3.13	A modification on a TrainBenchmark instance model . . . . .	39
3.14	Operation sequence on a distributed Rete net . . . . .	40
4.1	TrainBenchmark results measured on a single node . . . . .	43
4.2	The benchmark scenario. The green arrows and components are specific to the incremental scenario. . . . .	45
4.3	The non-incremental "batch" scenario benchmark's setup . . . . .	45

4.4	Execution times for load and first validation . . . . .	47
4.5	Execution times for transformation . . . . .	48
4.6	Execution times of the revalidation . . . . .	48
4.7	Total execution times for 50 validations . . . . .	49
4.8	Total execution times for 50 validations . . . . .	51
4.9	Execution times for transformation and revalidation . . . . .	51
A.1	An example graph based on the railway system metamodel . . . . .	64

# Bibliography

- [1] OpenLink Software: Virtuoso Universal Server. <http://virtuoso.openlinksw.com/>.
- [2] Sesame: RDF API and Query Engine. <http://www.openrdf.org/>.
- [3] EMF-IncQuery. <https://viastra.inf.mit.bme.hu/incquery/documentation>, October 2012.
- [4] Xtext. <http://www.eclipse.org/Xtext/>, October 2012.
- [5] 2013: What's Coming Next in Neo4j! <http://blog.neo4j.org/2013/01/2013-whats-coming-next-in-neo4j.html>, January 2013.
- [6] 4store. <http://4store.org/>, October 2013.
- [7] About Data Partitioning in Cassandra. [http://www.datastax.com/docs/1.1/cluster\\_architecture/partitioning](http://www.datastax.com/docs/1.1/cluster_architecture/partitioning), October 2013.
- [8] Akka. <http://akka.io/>, May 2013.
- [9] Apache Cassandra. <http://cassandra.apache.org/>, May 2013.
- [10] Apache Hadoop. <http://hadoop.apache.org/>, May 2013.
- [11] Apache Thrift. <http://thrift.apache.org/>, October 2013.
- [12] Blueprints. <http://blueprints.tinkerpop.com/>, May 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] Lustre. <http://lustre.org/>, October 2013.
- [16] Neoclipse. <https://github.com/neo4j/neoclipse>, May 2013.

- [17] NoSQL Databases. <http://nosql-database.org/>, May 2013.
- [18] Objectivity – InfiniteGraph. <http://www.objectivity.com/infinitegraph>, October 2013.
- [19] OrientDB Graph-Document NoSQL DBMS. <http://www.orientdb.org/>, October 2013.
- [20] Planet Cassandra – Companies. <http://planetcassandra.org/Company/ViewCompany>, October 2013.
- [21] Protocol Buffers – Google’s data interchange format. <https://code.google.com/p/protobuf/>, October 2013.
- [22] Sparsity-technologies: DEX high-performance graph database. <http://www.sparsity-technologies.com/dex>, October 2013.
- [23] The GraphML File Format. <http://graphml.graphdrawing.org/>, October 2013.
- [24] The Innovative Zing JVM. [http://www.azulsystems.com/sites/default/files/images/Innovative\\_Zing\\_JVM\\_v2.pdf](http://www.azulsystems.com/sites/default/files/images/Innovative_Zing_JVM_v2.pdf), August 2013.
- [25] The R Project for Statistical Computing. <http://www.r-project.org/>, October 2013.
- [26] TinkerPop. <http://www.tinkerpop.com/>, May 2013.
- [27] Acharya, A. et al. Implementation of production systems on message-passing computers. *IEEE Trans. Parallel Distr. Syst.*, 3(4):477–487, July 1992.
- [28] Don Batory. The LEAPS Algorithm. Technical report, Austin, TX, USA, 1994.
- [29] 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](http://mit.bme.hu/~rath/pub/theses/diploma_bergmann.pdf), 2008.
- [30] Gábor Bergmann. *Incremental Model Queries in Model-Driven Design*. Ph.d. dissertation, Budapest University of Technology and Economics, Budapest, 10/2013 2013.
- [31] Bergmann, Gábor et al. Incremental evaluation of model queries over EMF models. In *MODELS*, volume 6394 of *LNCS*. Springer, 2010.
- [32] Bin Cao, Jianwei Yin, Qi Zhang, and Yanming Ye. A MapReduce-Based Architecture for Rule Matching in Production System. In *CLOUDCOM*. IEEE, 2010.

- [33] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.
- [34] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
- [35] Csikós Donát. Incremental dependency analysis of a large software infrastructure. Master’s thesis, Budapest University of Technology and Economics, 2013.
- [36] Eclipse MDT Project. Eclipse OCL, 2011. <http://eclipse.org/modeling/mdt/?project=ocl>.
- [37] Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligences*, 19(1):17–37, 1982.
- [38] 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.
- [39] Thomas Goldschmidt and Axel Uhl. Efficient OCL impact analysis, 2011.
- [40] RDF Core Working Group. Resource Description Framework (RDF). <http://www.w3.org/RDF/>, 2004.
- [41] S. Harris, N. Lamb, and N. Shadbolt. 4store: The Design and Implementation of a Clustered RDF Store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, 2009.
- [42] 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%.
- [43] 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.
- [44] 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.

- [45] 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.
- [46] 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.
- [47] 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.
- [48] Benedek Izsó, Zoltán Szatmári, and István Ráth. High performance queries and their novel applications, 2012.
- [49] Adam Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):36–44, August 2009.
- [50] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.
- [51] 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.
- [52] 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.
- [53] Miranker, Daniel P et al. Diamond: A SPARQL query engine, for linked data based on the Rete match. *AIWD*, 2012.
- [54] Neo Technology. Neo4j. <http://neo4j.org/>, 2013.
- [55] Mikko Rinne. SPARQL update for complex event processing. In *ISWC'12*, volume 7650 of *LNCS*. 2012.
- [56] Marko A. Rodriguez and Peter Neubauer. Constructions from dots and lines. *CoRR*, abs/1006.2361, 2010.

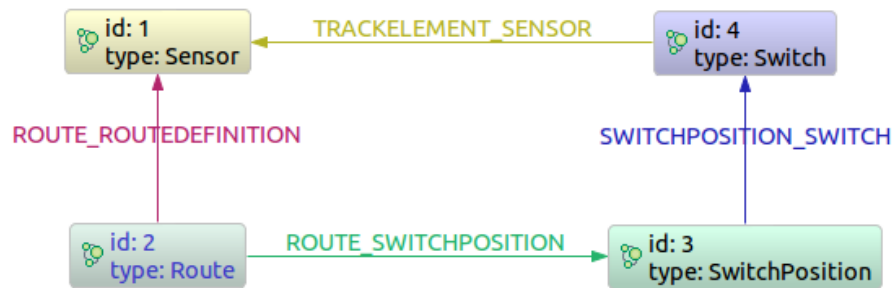
- [57] Markus Scheidgen. How Big are Models – An Estimation. Technical report, Department of Computer Science, Humboldt Universität zu Berlin, 2012.
- [58] Sherif Sakr. Supply cloud-level data scalability with NoSQL databases. <http://www.ibm.com/developerworks/cloud/library/cl-nosqldatabase/index.html>, March 2013.
- [59] Andrés Taylor and Alistair Jones. Cypher Query Language. <http://www.slideshare.net/graphdevroom/cypher-query-language>, 2012.
- [60] The Eclipse Project. Eclipse Modeling Framework. <http://www.eclipse.org/emf>, October 2012.
- [61] W3C. SPARQL Query Language for RDF. <http://www.w3.org/TR/rdf-sparql-query/>.



# Appendix A

## Graph Formats

In this chapter, we provide examples for the different graph 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 [23] 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.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <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">
3   <key id="type" for="node" attr.name="type" attr.type="string" />
4   <graph id="G" edgedefault="directed">
5     <node id="1">
6       <data key="type">Sensor</data>
7     </node>
8     <node id="2">
9       <data key="type">Route</data>
10    </node>
```

```

11 <node id="3">
12   <data key="type">SwitchPosition</data>
13 </node>
14 <node id="4">
15   <data key="type">Switch</data>
16 </node>
17 <edge id="0" source="2" target="1" label="ROUTE_ROUTEDEFINITION" />
18 <edge id="1" source="2" target="3" label="ROUTE_SWITCHPOSITION" />
19 <edge id="2" source="3" target="4" label="SWITCHPOSITION_SWITCH" />
20 <edge id="3" source="4" target="1" label="TRACKELEMENT_SENSOR" />
21 </graph>
22 </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.

```

1 {
2   "vertices": [
3     {
4       "type": "Sensor",
5       "_id": 1,
6       "_type": "vertex"
7     },
8     {
9       "type": "Route",
10      "_id": 2,
11      "_type": "vertex"
12    },
13    {
14      "type": "SwitchPosition",
15      "_id": 3,
16      "_type": "vertex"
17    },
18    {
19      "type": "Switch",
20      "_id": 4,
21      "_type": "vertex"
22    }
23  ],
24  "edges": [
25    {
26      "_id": 0,
27      "_type": "edge",
28      "_outV": 2,
29      "_inV": 1,
30      "_label": "ROUTE_ROUTEDEFINITION"
31    },
32    {
33      "_id": 1,
34      "_type": "edge",
35      "_outV": 2,

```

```

36     "_inV":3,
37     "_label":"ROUTE_SWITCHPOSITION"
38 },
39 {
40     "_id":2,
41     "_type":"edge",
42     "_outV":3,
43     "_inV":4,
44     "_label":"SWITCHPOSITION_SWITCH"
45 },
46 {
47     "_id":3,
48     "_type":"edge",
49     "_outV":4,
50     "_inV":1,
51     "_label":"TRACKELEMENT_SENSOR"
52 }
53 ]
54 }

```

**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 split-  
ted to blocks efficiently and processed on Hadoop nodes in a parallel fashion.

```

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

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <rdf:RDF
3   xmlns="http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#"
4   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
5   xmlns:swrl="http://www.w3.org/2003/11/swrl#"
6   xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"

```

```

7   xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
8   xmlns:owl="http://www.w3.org/2002/07/owl#"
9   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
10
11  <rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl">
12    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Ontology"/>
13  </rdf:Description>
14
15  <rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#Segment">
16    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
17    <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#Trackelement"/>
18  </rdf:Description>
19
20  <rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#Switch">
21    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
22    <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#Trackelement"/>
23  </rdf:Description>
24
25  <rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#1">
26    <rdf:type rdf:resource="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#Sensor"/>
27  </rdf:Description>
28
29  <rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#2">
30    <rdf:type rdf:resource="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#Route"/>
31  </rdf:Description>
32
33  <rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#3">
34    <rdf:type rdf:resource="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#Switch"/>
35  </rdf:Description>
36
37  <rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#4">
38    <rdf:type rdf:resource="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#SwitchPosition"/>
39  </rdf:Description>
40
41  <rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#3">
42    <TrackElement_sensor rdf:resource="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#1"/>
43  </rdf:Description>
44
45  <rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#4">
46    <SwitchPosition_switch rdf:resource="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#3"/>
47  </rdf:Description>

```

```
48
49 <rdf:Description rdf:about="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#2">
50   <Route_routeDefinition rdf:resource="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#1"/>
51   <Route_switchPosition rdf:resource="http://www.semanticweb.org/ontologies/2011/1/
    TrainRequirementOntology.owl#4"/>
52 </rdf:Description>
53
54 </rdf:RDF>
```

**Listing A.4:** *A graph based on the railway system metamodel stored in RDF format*