Project Number 611125

Technical Report – Train Benchmark

Version 0.11
2 July 2015
Draft

Public Distribution

BME

Project Partners: ARMINES, Autonomous University of Madrid, BME, IKERLAN, Soft-Maint, SOFTEAM, The Open Group, UNINOVA, University of York

Project Partner Contact Information

ARMINES Massimo Tisi Rue Alfred Kastler 4 44070 Nantes Cedex, France Tel: +33 2 51 85 82 09 E-mail: massimo.tisi@mines-nantes.fr	**Autonomous University of Madrid** Juan de Lara Calle Einstein 3 28049 Madrid, Spain Tel: +34 91 497 22 77 E-mail: juan.delara@uam.es
BME Daniel Varro Magyar Tudosok korutja 2 1117 Budapest, Hungary Tel: +36 146 33598 E-mail: varro@mit.bme.hu	**IKERLAN** Salvador Trujillo Paseo J.M. Arizmendiarrieta 2 20500 Mondragon, Spain Tel: +34 943 712 400 E-mail: strujillo@ikerlan.es
Soft-Maint Vincent Hanniet Rue du Chateau de L'Eraudiere 4 44300 Nantes, France Tel: +33 149 931 345 E-mail: vhanniet@sodifrance.fr	**SOFTEAM** Alessandra Bagnato Avenue Victor Hugo 21 75016 Paris, France Tel: +33 1 30 12 16 60 E-mail: alessandra.bagnato@softeam.fr
The Open Group Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org	**UNINOVA** Pedro Maló Campus da FCT/UNL, Monte de Caparica 2829-516 Caparica, Portugal Tel: +351 212 947883 E-mail: pmm@uninova.pt
University of York Dimitris Kolovos Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 32516 E-mail: dimitris.kolovos@york.ac.uk	

Contents

Document Control

Version	Status	Date
0.1	Document outline	30 November 2013
0.2	First draft - structure ready	17 January 2014
0.3	Second draft - incorporation of ASE material	06 March 2014
0.5	Third draft - revised contents	10 March 2014
0.8	Fourth draft - results complete, analysis needs rework	10 April 2014
0.9	Fifth draft - complete	1 June 2014
0.10	Sixth draft - revisions in scenarios and transformations	15 March 2015

Executive Summary

The main goal of the Train Benchmark is to measure the *execution time* of graph-based query processing tools with particular emphasis on incremental query reevaluation. The execution time is measured against models of growing sizes generated by an instance model generator. This way, the scalability of the tools is evaluated. The Train Benchmark also demonstrates other abilities of the tools, including transformation capabilities, conciseness of the query and transformation language, convenience of the API, compatibility with different metamodeling languages and so on. The Train Benchmark is a general benchmarking framework, that contains a specific benchmark test case set built around a metamodel inspired by railway system design. This test case set also comes with a set of queries and transformation operations.

The current tech report includes information on two variants of the Train Benchmark: the original version was described in [30] and describes the incremental well-formedness checking scenario. The extended version was described in [21] and introduces query and instance model query metrics to quantitatively assess the "difficulty" of various model query-instance model combinations to find those metrics that are best for predicting the query evaluation time without running the query itself.

The most up-to-date supplementary material is found at `https://github.com/FTSRG/trainbenchmark-docs`

Chapter 1

Overview

Scalability issues in model-driven engineering arise due to the increasing complexity of modeling workloads. This complexity comes from two main factors: (i) *instance model sizes* are exhibiting a tremendous growth as the complexity of systems-under-design is increasing, (ii) increasing *feature sophistication* in toolchains, such as complex model validation or transformations.

One of the the most computationally expensive tasks in modeling applications are *model queries*. While there are a number of existing benchmarks for queries over relational databases [29] or graph stores [13, 26], modeling tool workloads are significantly different. Specifically, modeling tools use much more complex queries than typical transactional systems, and the real world performance is more affected by response time (i.e. execution time for a specific operation such as validation or transformation) than throughput (i.e. the amount of parallel transactions).

1.1 Overview

To address this challenge, the Train Benchmark [30, 6] is a macro benchmark that aims to measure batch and incremental query evaluation performance, in a scenario that is specifically modeled after *model validation* in (domain-specific) modeling tools: at first, the entire model is validated, then after each model manipulation (e.g. the deletion of a reference) is followed by an immediate re-validation. The benchmark measures execution times for four phases: (1) *read* (2) *check* (3) *edit* (4) *re-check*.

The Train Benchmark computes two derived results based on the recorded data: (1) *batch validation time* (the sum of the *read* and *check* phases) represents the time that the user must wait to start to use the tool; (2) *incremental validation time* consists of the *edit* and *re-check* phases performed 100 times, representing the time that the user spent waiting for the tool validation.

1.2 Instance Models

The Train Benchmark uses a domain-specific model of a railway system that originates from the MOGENTES EU FP7 [5] project, where both the metamodel and the well-formedness rules were defined by railway domain experts. This domain enables the definition of both simple and more complex model queries while it is simple enough to incorporate solutions from other technological spaces (e.g. ontologies, relational databases and RDF). This allows the comparison of the performance aspects of wider range of query tools from a constraint validation viewpoint.

The instance models are systematically generated based on the metamodel and the defined complex model queries: small instance model fragments are generated based on the queries, then they are placed, randomized and connected to each other. The methodology takes care of controlling the number of matches of all defined model queries. To break symmetry, the exact number of elements and cardinalities are randomized.

This brings artificially generated models *closer to real world instances* and *prevents query tools from efficiently storing* or caching of instance models. During the generation of the railway system model, errors are injected at random positions. These errors can be found in the check phase of the benchmark, which are reported and can be corrected during the edit phase. The initial number of constraint violating elements are low (<1% of total elements).

1.3 Queries and Transformations

Queries are defined informally in plain text (in a tool independent way) and also formalized using the standard OCL language as a reference implementation (available on the benchmark website [6]). The queries range from simple attribute value checks to complex navigation operations consisting of several (4+) joins.

The functionally equivalent variants of these queries are formalized using the query language of different tools applying tool based optimizations. As a result, all query implementations must return (the same set of) invalid instance model elements.

In the *edit* phase, the model is modified to change the result set to be returned by the query in the *recheck* phase. For simulating manual modifications, the benchmark always performs 100 random edits (fixed low constant) which increases the number of erroneous elements. An edit operation only modifies a single model elements at once – more complex model manipulation is modelled as series of edits.

1.4 Evaluation of Measurements

The Train Benchmark defines a Java-based framework and application programming interface that enables the integration of additional metamodels, instance models, query implementations and even new benchmark scenarios (which may be different from the original four-phase concept). The default implementation contains a benchmark suite for queries implemented in Java, Eclipse OCL and EMF-INCQUERY.

Measurements are recorded automatically in a machine-processable format (CSV) that is automatically processed by R [7] scripts. An extended version of the Train Benchmark [21] features several (instance model, query-specific and combined) *metrics* that can be used to characterize the “difficulty” of benchmark cases numerically, and – since they can be evaluated automatically for other domain/model/query combinations – allow to compare the benchmark cases with other real-world workloads.

Chapter 2

Train Benchmark Technical Specification

This chapter discusses the technical specification of the Train Benchmark. In order to ensure fair comparison of different tools, the specification describes the queries, transformations and the instance model generator in detail. The phases in the benchmark are also strictly defined.

Currently, the Train Benchmark comes in two versions: the original version (Section 2.1) and an extended version (Section 2.2), first discussed in paper [21].

2.1 Original Version

A *test case* configuration for every tool consists of an *instance model* (Section 2.1.6) with an *instance model size*, a *predefined query* (Section 2.1.4) describing constraint violating elements and the name of the *scenario* (Section 2.1.2) to run.

As a result of a testcase run, the *execution times* of each phase, the *memory usage* and the *number of erroneous elements* are measured and recorded. The number of invalid elements are used to check the correctness of the validation, however the set of element identifiers must also be available for later processing.

2.1.1 Phases

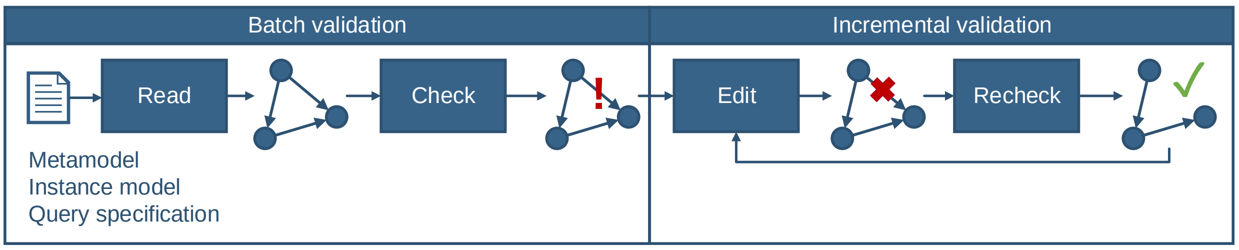


Figure 2.1: The phases of the benchmark.

To measure performance for re-validating a model after modifications, four benchmark *phases* were defined, as illustrated in Figure 2.1.

1. During the *read* phase, the previously generated instance model and validation query are loaded from hard drive to memory. This includes the parsing of the input as well as initializing data structures of the tool.

2. In the *check* phase, the instance model is queried to identify invalid elements. This can be as simple as reading the results from cache, or the model can be traversed based on some index. By the end of this phase, erroneous objects need to made available in a collection for further processing.

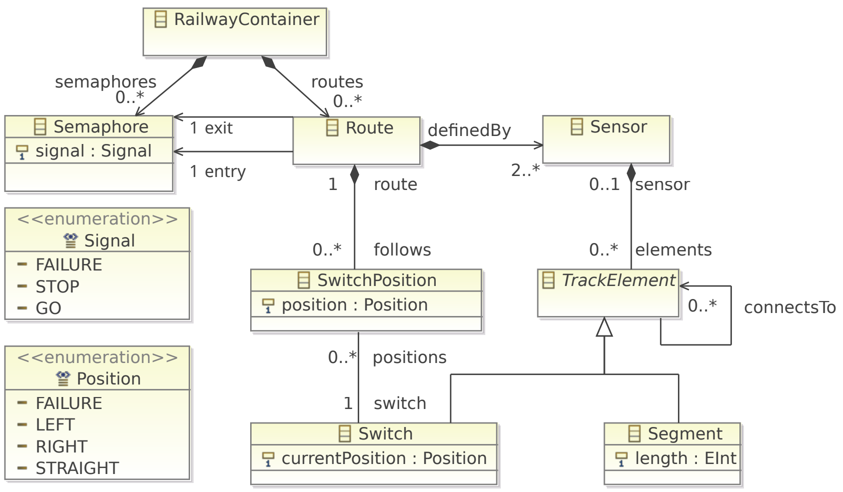
3. In the *edit* phase, the model is changed to simulate effects (and measure performance) of model modifying operations. At the beginning of this phase "query-like" functions are used to gather elements to be modified, however this time is excluded, and only the required time of model editing operations are recorded in this phase. (As query performance is measured in the check phases.)

4. The re-validation of the model is carried out in the *re-check* phase similarly to the *check* phase.

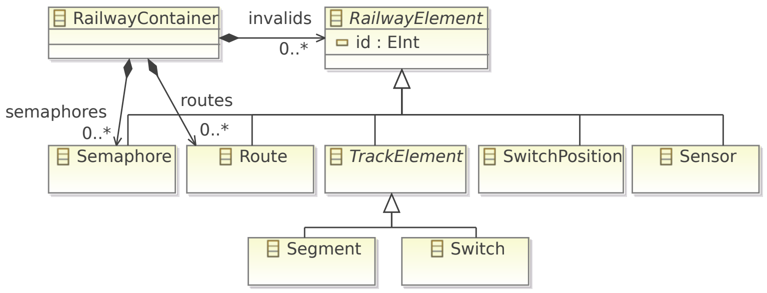
2.1.2 Use Case Scenarios of the Benchmark

Paper [10] analyzes performance of algorithms used for graph pattern query evaluation and identifies two use cases where efficient *incremental model validation* is required. The batch part of our benchmark consists of the execution of the read and first check phases. Inspired by paper [10], three *scenarios* were defined to measure different use cases:

- *Batch validation scenario* (Batch): In this scenario the model is *read* in one batch from storage, than a model validation is carried out by executing the query in the check phase. Such use case is performed when a model editor is opened and initial validations are issued by the designer.
- *Fault injection scenario* (Inject): The fault injection scenario extends the batch validation scenario by differential model modification and re-checking phases. After the batch validation a small model manipulation step is performed (e.g. a reference is deleted), which is immediately followed by re-validation to get instantaneous feedback. In this scenario such small edit and re-check phases are executed in sequence.
  Such scenario occurs when someone uses a typical UML editor (for designing software solutions), or a domain-specific editor where elements or relations are added one-by-one. These editors should detect design errors quickly and early in the development process to let engineers refine models and cut down debugging and error correction costs.
- *Automated model repair scenario* (Repair): The automated model repair scenario extends the batch validation scenario by differential model modification and re-checking phases. In the edit phase, the model is repaired, based on the erroneous objects identified during the batch validation. This is carried out by the tool itself performing mass edits automatically. Finally, the whole model is re-checked, and remaining or newly introduced errors are reported.



(a) Containment hierarchy and references



(b) Supertype relations

Figure 2.2: The metamodel of the Train Benchmark.

Efficient execution of such a use case is necessary during refactoring, incremental code generation, or when a model is transformed from a source language to a target language by a model transformation program, using model synchronisation.

2.1.3 Metamodel

The metamodel of the railway domain used in the Train Benchmark is depicted in Figure 2.2. A train route can be defined by a set of sensors. Sensors are associated with track elements: track segments (with a specific length) or switches. A route may have associated switch positions which describe the required state of a switch belonging to the route. Different route definitions can specify different states for a specific switch.

2.1.4 Queries

In the validation and re-validation phases of the benchmark, the queries return the elements violating the well-formedness constraint defined by the test case. These constraints are first defined informally in plain text and then formalized using a query language suited for the benchmarked tool. As a result, the query must return a set of the invalid instance model elements' identifiers.

Two simple queries involving maximum 2 objects (PosLength and SwitchSensor) and four complex queries involving 4–8 objects and multiple join operations (ConnectedSegments, RouteSensor, SemaphoreNeighbor and SwitchSet) were defined. Simple queries are able to quickly distinguish efficient tools from inefficient ones, while complex queries are used to differentiate faster model query technologies.

In the following, we present the queries defined in the Train Benchmark. We describe the semantics and the goal of each query. We also show the associated graph pattern and relational algebra query.

Relational Schemas

For formulating the queries in relational algebra we define the following relational schemas for representing the vertices (objects) in the graph (instance model).

- $\mathit{Route}\left(\mathit{id}\right)$
- $\mathit{Segment}\left(\mathit{id}, \mathit{length}\right)$
- $\mathit{Semaphore}\left(\mathit{id}, \mathit{signal}\right)$
- $\mathit{Sensor}\left(\mathit{id}\right)$
- $\mathit{Switch}\left(\mathit{id}, \mathit{currentPosition}\right)$
- $\mathit{SwitchPosition}\left(\mathit{id}, \mathit{position}\right)$
- $\mathit{TrackElement}\left(\mathit{id}\right)$

The edges[1] are represented with the following relational schemas:

- $\mathit{entry}\left(\mathit{Route}, \mathit{Semaphore}\right)$
- $\mathit{exit}\left(\mathit{Route}, \mathit{Semaphore}\right)$
- $\mathit{follows}\left(\mathit{Route}, \mathit{SwitchPosition}\right)$
- $\mathit{definedBy}\left(\mathit{Route}, \mathit{Sensor}\right)$
- $\mathit{switch}\left(\mathit{SwitchPosition}, \mathit{Switch}\right)$
- $\mathit{sensor}\left(\mathit{Switch}, \mathit{Sensor}\right)$
- $\mathit{connectsTo}\left(\mathit{TrackElement}_1, \mathit{TrackElement}_2\right)$

Graph Patterns

In the following, we represent the graph pattern representation for the violation of each well-formedness constraint. For defining the patterns and transformations, we used a graphical syntax similar to GROOVE [25] with a couple of additions:

- The invalid model elements returned by the query are shown in **bold** font.
- Filter conditions are shown in *italic* font.
- Negative application conditions are shown with in a red rectangle with the NEG caption.
- The insertions are with a «new» caption, while deletions are marked with a «del» caption. Attribute updates are also show in green.

[1]also called relationships or references in various implementations

ConnectedSegments

Description. The ConnectedSegments well-formedness constraint requires that each sensor must have at most five segments attached to it. Therefore, the query (Figure 2.4) checks for sensors that have at least six segments attached to them. The SPARQL representation of the query is shown in Figure 2.3.

Goal. The query checks for „chains” similar to a transitive closure.

```
SELECT DISTINCT ?sensor ?segment1 ?segment2 ?segment3 ?segment4 ?segment5 ?segment6
WHERE
{
  ?segment1 base:sensor ?sensor .
  ?segment1 base:connectsTo ?segment2 .
  ?segment2 base:connectsTo ?segment3 .
  ?segment3 base:connectsTo ?segment4 .
  ?segment4 base:connectsTo ?segment5 .
  ?segment5 base:connectsTo ?segment6 .
  ?segment6 base:sensor ?sensor .

  ?sensor rdf:type base:Sensor .
  ?segment1 rdf:type base:Segment .
  ?segment2 rdf:type base:Segment .
  ?segment3 rdf:type base:Segment .
  ?segment4 rdf:type base:Segment .
  ?segment5 rdf:type base:Segment .
  ?segment6 rdf:type base:Segment .
}
```

Figure 2.3: ConnectedSegments query in SPARQL.

Relational algebraic form. The ConnectedSegments query can be formalized in relational algebra as:

$$\begin{aligned}
&\pi_{id}\sigma_{Sensor_1.id=Sensor_2.id}\big(Sensor_1\\
&\quad\bowtie_{Sensor_1.id=connectsTo_1.TrackElement_1}\ connectsTo_1\\
&\quad\bowtie_{connectsTo_1.TrackElement_2=connectsTo_2.TrackElement_1}\ connectsTo_2\\
&\quad\bowtie_{connectsTo_2.TrackElement_2=connectsTo_3.TrackElement_1}\ connectsTo_3\\
&\quad\bowtie_{connectsTo_3.TrackElement_2=connectsTo_4.TrackElement_1}\ connectsTo_4\\
&\quad\bowtie_{connectsTo_4.TrackElement_2=connectsTo_5.TrackElement_1}\ connectsTo_5\\
&\quad\bowtie_{connectsTo_5.TrackElement_2=Sensor_2.id}\ Sensor_2\\
&\big)
\end{aligned}$$

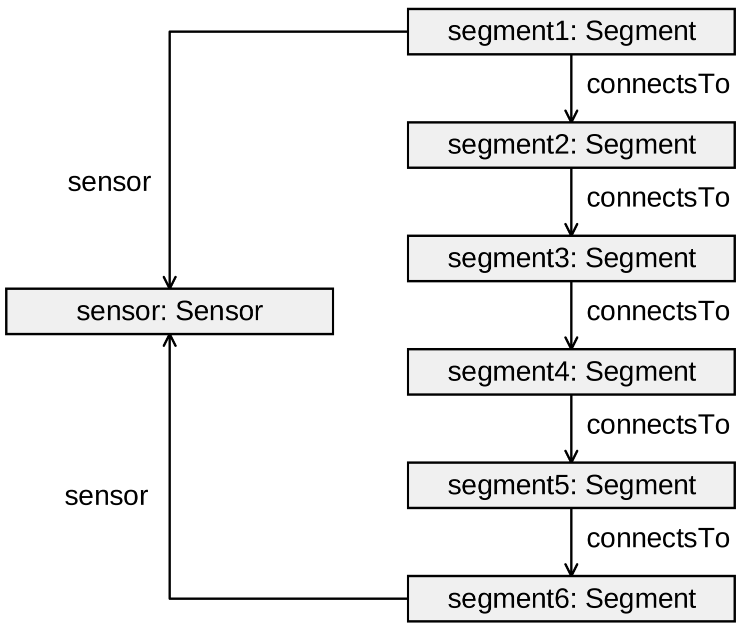


Figure 2.4: The ConnectedSegments pattern.

PosLength

Description. The PosLength well-formedness constraint requires that a segment must have positive length. Therefore, the query (Figure 2.6) checks for segments with a length less than or equal to zero. The SPARQL representation of the query is shown in Figure 2.5.

Goal. The query checks whether an object has an attribute. If it does, the value is checked. Checking attributes is a real world use case, even if a very simple one. Note that simple string checking is also measured in the Berlin SPARQL Benchmark [13] and it concludes that for most tools the string comparison algorithm dominates the query time.

```
SELECT ?segment ?length
WHERE
{
  ?segment rdf:type base:Segment .
  ?segment base:length ?length .

  FILTER (?length <= 0)
}
```

Figure 2.5: PosLength query in SPARQL.

Relational algebraic form. The PosLength query can be formalized in relational algebra as:

$$\pi_{id}\left(\sigma_{length \leq 0}\left(Segment\right)\right)$$

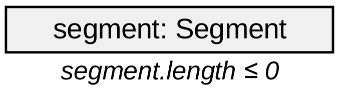


Figure 2.6: The PosLength pattern.

RouteSensor

Description. The RouteSensor well-formedness constraint requires that all sensors that are associated with a switch that belongs to a route must also be associated directly with the same route. Therefore, the query (Figure 2.9) looks for sensors that are connected to a switch, but the sensor and the switch are not connected to the same route. The SPARQL representation of the query is shown in Figure 2.7.

Goal. This pattern checks for the absence of circles, so the efficiency of the join and the antijoin operations is tested.

```
OPTIONAL {
  ?xRoute ?xRouteDefinition ?xSensor .
  FILTER (sameTerm(base:routeDefinition,
                ?xRouteDefinition))
} .
FILTER (!bound(?xRouteDefinition))
```

Figure 2.7: The RouteSensor query in SPARQL 1.1.

Remark. Note that the negative application condition (NAC) part (`FILTER NOT EXISTS`) is a SPARQL 1.1 feature. In SPARQL 1.0, we have to use the approach shown in Figure 2.8.

```
SELECT DISTINCT ?route ?sensor ?swP ?sw
WHERE
{
  ?route base:follows ?swP .
  ?swP base:switch ?sw .
  ?sw base:sensor ?sensor .

  ?route rdf:type base:Route .
  ?swP rdf:type base:SwitchPosition .
  ?sw rdf:type base:Switch .
  ?sensor rdf:type base:Sensor .

  OPTIONAL {
    ?route ?xDefinedBy ?sensor .
    FILTER (sameTerm(base:definedBy, ?xDefinedBy))
  } .
  FILTER (!bound(?xDefinedBy))
}
```

Figure 2.8: The NAC of the RouteSensor pattern in SPARQL 1.0.

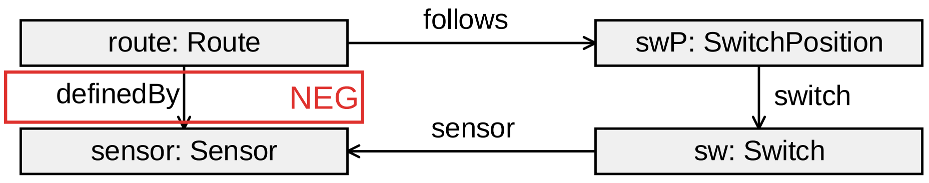


Figure 2.9: The RouteSensor pattern.

Relational algebraic form. The RouteSensor query can be formalized in relational algebra as:

$$\pi_{Route}\left(follows \bowtie switch \bowtie sensor \triangleright definedBy\right)$$

SemaphoreNeighbor

Description. The SemaphoreNeighbor well-formedness constraint requires that the routes that are connected through sensors and track elements have to belong to the same semaphore. Therefore, the query (Figure 2.11) checks for routes which have an exit semaphore and a sensor connected to another sensor (which is in a definition of another route) by two track elements, but there is no other route that connects the same semaphore and the other sensor. The SPARQL representation of the query is shown in Figure 2.10.

Goal. This pattern checks for the absence of circles, so the efficiency of the join operation is tested. One-way navigable references are also present in the constraint, so the efficient evaluation of these are also tested. Subsumption inference is required, as the two track elements can be switches or segments.

```
SELECT DISTINCT ?semaphore ?route1 ?route2 ?sensor1 ?sensor2 ?te1 ?te2
WHERE
{
  ?route1 base:exit ?semaphore .
  ?route1 base:definedBy ?sensor1 .
  ?te1 base:sensor ?sensor1 .
  ?te1 base:connectsTo ?te2 .
  ?te2 base:sensor ?sensor2 .
  ?route2 base:definedBy ?sensor2 .

  FILTER ( ?route1 != ?route2 ) .

  OPTIONAL {
    ?route2 ?entry ?semaphore .
    FILTER (sameTerm(base:entry, ?entry))
  } .
  FILTER (!bound(?entry))
}
```

Figure 2.10: The SemaphoreNeighbor query in SPARQL.

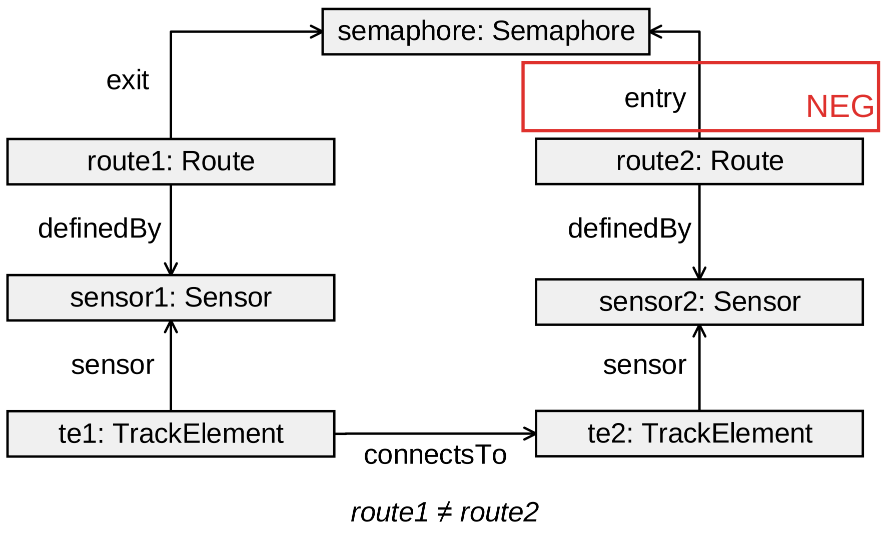


Figure 2.11: The SemaphoreNeighbor pattern.

Relational algebraic form. The SemaphoreNeighbor query can be formalized in relational algebra as:

$$\pi_{entry.Route}\Big(\sigma_{entry.Route \neq definedBy_2.Route}\Big(\\ entry \bowtie definedBy_1 \bowtie sensor_1 \bowtie \\ connectsTo \bowtie sensor_2 \bowtie definedBy_2 \triangleright \\ (exit \bowtie definedBy_3) \\ \Big)\Big)$$

SwitchSensor

Description. The SwitchSensor well-formedness constraint requires that every switch must have at least one sensor connected to it. Therefore, the query (Figure 2.13) checks for switches that have no sensors associated with them. The SPARQL representation of the query is shown in Figure 2.12.

Goal. This query checks whether an object is connected to a relation. This pattern is common in more complex queries, e.g. it is used in the RouteSensor and the SemaphoreNeighbor queries.

Relational algebraic form. The SwitchSensor query can be formalized in relational algebra as:

$$Switch \triangleright Sensor$$

```
SELECT DISTINCT ?xSwitch WHERE {
 ?xSwitch rdf:type base:Switch .

 FILTER NOT EXISTS {
   ?xSensor rdf:type base:Sensor .
   ?xSwitch base:sensor ?xSensor .
 } .
}
```

Figure 2.12: The SwitchSensor query in SPARQL.

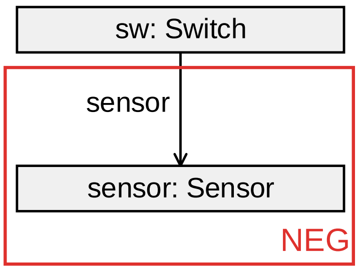


Figure 2.13: The SwitchSensor pattern.

SwitchSet

Description. The entry semaphore of a route may only show GO if all switches along the route are in the position prescribed by the route. The SPARQL representation of the query is shown in Figure 2.14.

Goal. This pattern tests the efficiency of the join and filtering operations.

```
SELECT DISTINCT ?semaphore ?route ?swP ?sw ?currentPosition ?position
WHERE
{
   ?route base:entry ?semaphore .
   ?route base:follows ?swP .
   ?swP base:switch ?sw .
   ?semaphore base:signal base:SIGNAL_GO .
   ?sw base:currentPosition ?currentPosition .
   ?swP base:position ?position .

   ?semaphore rdf:type base:Semaphore .
   ?route rdf:type base:Route .
   ?swP rdf:type base:SwitchPosition .
   ?sw rdf:type base:Switch .

   FILTER (!sameTerm(?currentPosition, ?position))
}
```

Figure 2.14: The SwitchSet query in SPARQL.

Relational algebraic form. The SwitchSet query can be formalized in relational algebra as:

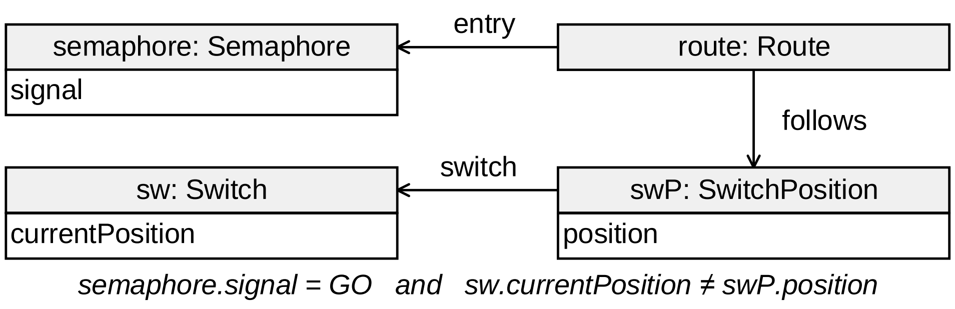


Figure 2.15: The SwitchSet pattern.

$$
\begin{aligned}
&\pi_{id}\sigma_{Semaphore.signal=\text{"GO"}\wedge Switch.currentPosition\neq SwitchPosition.position}\Big(Route\\
&\quad\bowtie_{Route.entry=Semaphore.id} Semaphore\\
&\quad\bowtie_{Route.id=follows.Route} follows\\
&\quad\bowtie_{follows.SwitchPosition=SwitchPosition.id} SwitchPosition\\
&\quad\bowtie_{SwitchPosition.id=switch.SwitchPosition} switch\\
&\quad\bowtie_{switch.Switch=Switch.id} Switch\\
&\Big)
\end{aligned}
$$

2.1.5 Transformations

In the *edit* phase the model is modified to change the result set returned in the succeeding re-check phase.

The Train Benchmark defines model transformations for each query. The transformations are also represented as graph transformations. The insertions are shown in green with a «new» caption, while deletions are marked with a red cross and a «del» caption. In general, the goal of these transformations is to remove a subset of the invalid elements from the model.

	Inject (modify = 10)				Repair (modify = Res1 × 10%)			
#Objects	**#Sensors**	**Result1**	**Modify RS**	**Result2**	**#Sensors**	**Result1**	**Modify RS**	**Result2**
6032	928	19	10	29	967	94	9	85
11710	1804	41	10	51	1936	193	19	174
23180	3575	68	10	76	3545	348	34	314
46728	7210	140	10	150	6691	642	64	578
87396	13465	264	10	274	13650	1301	130	1171
175754	27074	510	10	520	27190	2606	260	2346
354762	54653	1048	10	1058	55708	5324	532	4792
708770	109185	2071	10	2081	110291	10623	1062	9561
1415954	218140	4215	10	4224	219305	21097	2109	18988
2837336	437089	8501	10	8510	437025	41762	4176	37586

Table 2.1: Modification in the RouteSensor test case.

Table 2.1. shows the instance model characteristics and the effect of the modify phase in the RouteSensor case. The first column counts the number of instance model elements. The second and the fifth column show the number of Sensors in the model. The two scenarios process different instance models and modify them differently:

- In the *Inject scenario* (where a developer is assumed to sit in front of an editor) the initial number of constraint violating elements are low (0.3% of model elements for the RouteSensor case), so it can be understood and resolved by a user using the editor.
  During the modification the user always performs 10 random edits (fixed low constant) which *increase* the number of erroneous elements. These edit operations modify only some elements of the model and do not add or remove modules containing multiple instance model elements.
- In the *Repair scenario* the initial number of errors is higher (1.5% of objects for the RouteSensor case). This scenario models the case when these errors are (partially) processed by a model transformation program automatically.
  In the edit phase the program modifies 10% of the elements of the result set retrieved from the batch query. These modifications always correct an invalid element in the model, so the number of invalid elements *decreases* (see Table 2.2).

Table 2.2. displays the effect of model changes to the result set size and the type of edit operations (add, update, delete).

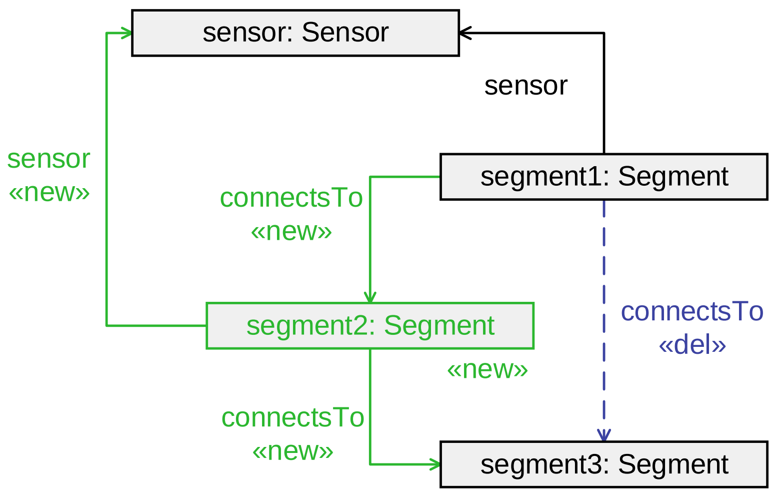
	Inject		**Repair**	
	Modification type	**RSS change**	**Modification type**	**RSS change**
ConnectedSegments	Insert and delete	Increase	Insert and delete	Decrease
PosLength	Update	Increase	Update	Decrease
RouteSensor	Delete	Increase	Delete	Decrease
SemaphoreNeighbor	Update	Increase	Update	Decrease
SwitchSensor	Delete	Increase	Add	Decrease
SwitchSet	Update	Increate	Update	Decrease

Table 2.2: Modification type for the queries.

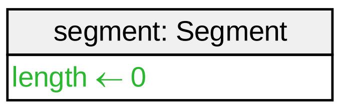
The modifications in more detail:

- *Inject scenario:* in this case elements to be edited are selected from the whole instance model, i.e. they may be valid or invalid.
  - *PosLength:* Randomly selected *segments' length* attribute is updated to 0, which means that an error is injected (Figure 2.16b).[2]
  - *RouteSensor:* The *definedBy* edges between the randomly selected *routes* and their *first* connected *sensor* are removed (at most one edge for each route, see Figure 2.16c).
  - *SemaphoreNeighbor:* Errors are introduced by disconnecting the *entry* edge of the selected *routes* (Figure 2.16d). (According to the metamodel, a *route* may only have 0 or 1 *entry* edges).

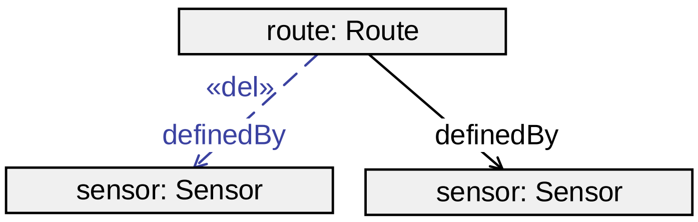
[2]In EMF this means that an `int` attribute is set (updated), while in other representations (e.g. RDF databases) first the assertion about the old value is removed and the assertion stating the new value of the length is inserted.



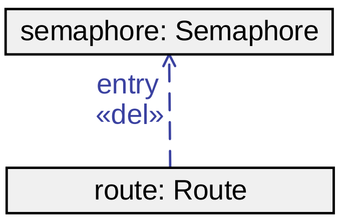
(a) ConnectedSegments



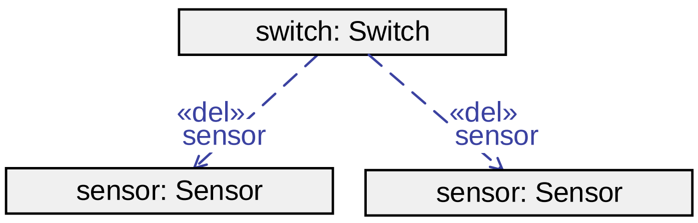
(b) PosLength



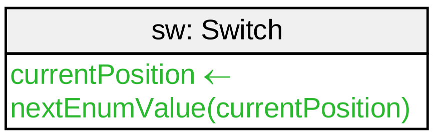
(c) RouteSensor



(d) SemaphoreNeighbor



(e) SwitchSensor



(f) SwitchSet

Figure 2.16: Transformations in the Inject scenario.

  - *SwitchSensor:* Errors are injected by randomly selecting *switches* and deleting all their edges to *sensors*. If the chosen *switch* was invalid, it did not have such an edge, so no edges are deleted and the *switch* stays invalid. If the chosen *switch* was valid, it will become invalid (Figure 2.16e).

- *Repair scenario:* in this case elements to be edited are selected from the result of the previous query. The transformations provide quick fix-like repair operations.

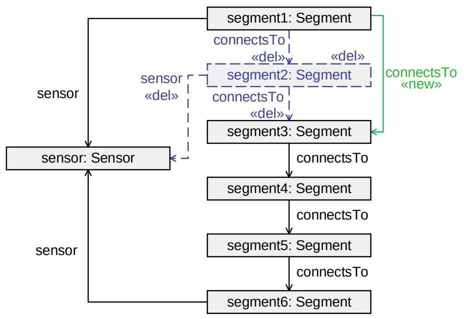
  - *PosLength:* Random elements are selected from the set of invalid *segments* and their values are updated to $-oldValue + 1$ (Figure 2.17b).
  - *RouteSensor:* Randomly selected invalid *sensors* are disconnected from the *switch*, which means that the constraint will no longer apply (Figure 2.17c).
  - *SemaphoreNeighbor:* Disconnect *exit* references of randomly selected invalid *routes*, resulting in a structure where the constraint must not hold for the actual route (Figure 2.17d).
  - *SwitchSensor:* Random elements are selected from the set of invalid *switches* and are connected to newly created *sensors*.
    In EMF this means the creation of a new Sensor which is added to the switch and also to the root container object. In ontology the triples asserting the connection between the new Sensor and the Switch, as well as its type are inserted into the knowledge base (Figure 2.17e).

2.1.6 Instance Model Generation

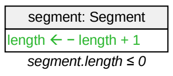
In the first phase of the benchmark, a previously generated *instance model* is loaded from the file system. These models are systematically generated based on the metamodel and on the model queries. Randomized instance model fragments are generated and connected to each other. The generation process takes care of controlling the number of matches for all model queries.

To break symmetry, the exact number of elements and cardinalities are randomized. This brings artificially generated models *closer to real world instances*, and *prevents query tools from this kind of efficient storing* of instance models. During the generation of the railway system model, errors are injected at random positions. The initial number of constraint violating elements that are low (below one percent of the total number of elements), and are deterministically placed, thanks to *pseudorandom* generation. These errors have to be found in the check phase of the benchmark, and can be corrected during the edit phase.

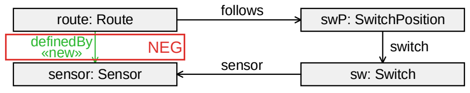
To show some characteristics of the generated instance models, the distribution of the object types and the average number of edges for each object are presented in Figure 2.2. In the case of classes the percentage of instances is shown: e.g. 3.4% of the model elements are instance of the class *Switch*, 77.0% are *Segment*s, thus 80.4% are *TrackElement*s. The average number of the given relation for an instance is displayed for associations: e.g. there are 9.5 *switchPosition* relations in average for every instance of the *Route* class.



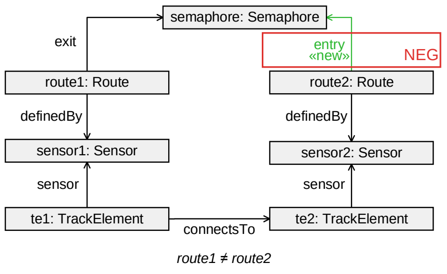
(a) ConnectedSegments



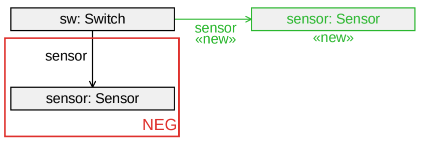
(b) PosLength



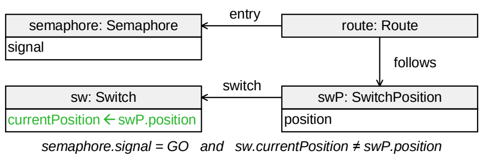
(c) RouteSensor



(d) SemaphoreNeighbor



(e) SwitchSensor



(f) SwitchSet

Figure 2.17: Transformations in the Repair scenario.

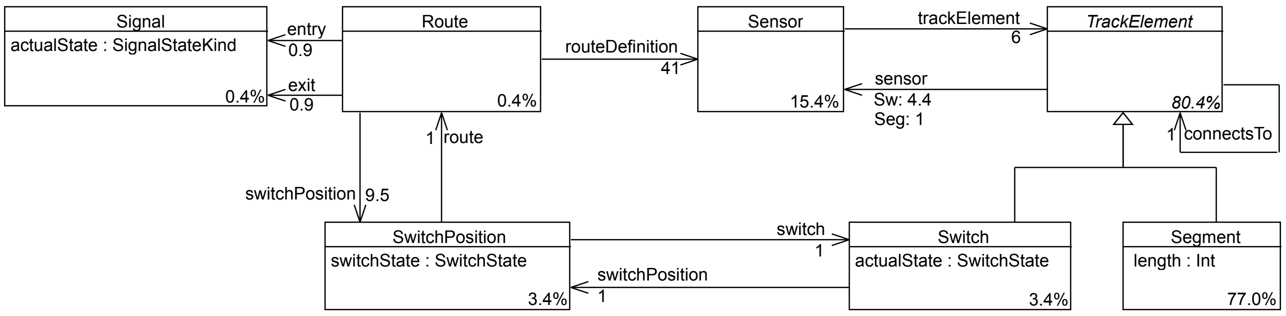


Figure 2.18: The railway metamodel and instance model characteristics.

2.2 Extended version

Section 2.1 describes a benchmark that can be used to compare performance of different tools. This Sections extends this benchmarking concept by characterizing instance models and queries by metrics, and evaluate which tools are the most sensitive to which metrics.

2.2.1 Overview

The aim of our metrics and benchmarking scenarios is to give a precise mechanism for identifying key factors in selecting between different query evaluation technologies.

The presented metrics for our evaluation (see in Section 2.2.2) were constructed based on a set of already and widely used metrics, extended with two more specific ones that aims to give a gross upper bound on the cost of query evaluation.

For the benchmark scenario we opted for a simple execution schema that represents a batch validation scenario. This is the same as the first two phase of the original benchmark. See Figure 2.1 and Section 2.1.1 for a description of the *read* and *check* phases.

2.2.2 Metrics

Our investigation relies on a selection of metrics that quantitatively describe the task of a certain query, independently of the actual strategy or technological solution that provides the query results. Broadly speaking, such a querying task consist of (a) an instance model, (b) a query specification that defines what results should be yielded, (c) a runtime context in which the queries are evaluated, such as the frequency of individual query evaluations and model manipulation inbetween.

The metrics discussed in the following, characterize instance models, queries or their combination, without characterizing a unique property of a specific graph/query description language or environment. Most of these metrics have previously been defined by other sources, while others are newly proposed in this paper.

Metrics for Instance Model Only

Clearly, properties of the instance model may have a direct effect on query performance, e.g. querying larger models may consume more resources.

A first model metric is model size, which can be defined either as the number of objects (metric `countNodes`), the number of references (edges) between objects (`countEdges`), the number of attribute value assignments (not used in the paper); or some combination of these three, such as their sum (`countTriples`), which is basically the total number of model elements / RDF triples. This is complemented by the number of different classes the objects in the model belong to (`countTypes`), and the instance count distribution of the classes. Additional important model metrics characterize the distribution of the out-degrees and in-degrees (the number of edges incident on an object), particularly the maximum and average degrees (`maxInDegree`, `maxOutDegree`, `avgInDegree` and `avgOutDegree`).

The metrics discussed above have been defined e.g. in [27], along with other metrics such as the relative frequencies of the edge label sequences of directed paths of length 2 or 3.

Metrics for Query Specification Only

The query specification is a decisive factor of performance as well, as complex queries may be costly to evaluate. Such query metrics can be defined separately, in several query formalisms. Due to the close analogies between graph patterns and SPARQL queries, we can consider these metrics applying to graph pattern-like queries in general. This allows us to formulate and calculate metrics expressed on the graph patterns interpreted by EMF-INCQUERY, and characterize the complexity of the equivalent SPARQL query with the same metric value.

As superficial metrics for graph pattern complexity, we propose the number of variables (`numVariables`) and the number of parameter variables (`numParameters`); the number of pattern edge constraints (`numEdgeConstraints`) and the number of attribute check constraints (`numAttrChecks`); finally the maximum depth of nesting NACs (`nestedNacDepth`). Some of these are similar/equivalent to SPARQL query metrics defined in [18]. Other metrics proposed by [18] are mostly aimed at measuring special properties relevant to certain implementation strategies.

Metrics for Combination of Query and Instance Model

The following two metrics (defined previously in literature) characterize the query and the instance model together.

The most trivial such metric is the cardinality of the query results (metric `countMatches`); intuitively, a query with a larger result set typically takes longer to evaluate on the same model, while a single query is typically more expensive to evaluate on models where it has more matches. The metric `selectivity` is proposed by [18], is the ratio of the number of results to the number of model elements (i.e. `countMatches/countTriples`).

New Metrics for Assessing Query Evaluation Difficulty

We propose two more metrics that take query and instance model characteristics into account. Our aim is to provide a gross upper bound on the cost of query evaluation. We consider all *enumerable* constraints in the query, for which it is possible to enumerate all tuples of variables satisfying it; thus edge constraints and pattern composition are enumerable, while NACs and attribute checks in general are not. At any given state of evaluation, a hypothetical search-based query engine has either already identified a single occurrence of an enumerable constraint c (e.g. a single instance of an edge type for the corresponding edge constraint), or not; there are therefore $|c| + 1$ possible cases for c, where $|c|$ is the number of different ways that c can be satisfied in the model. This gives $\prod_c 1 + |c|$ as the overestimate of the search space of the query evaluator. To make this astronomical figure manageable, we propose the absolute difficulty metric (`absDifficulty`) as the logarithm of the search space size, i.e. $\ln \prod_c (1 + |c|) = \sum_c \ln(1 + |c|)$.

The result size is a lower bound of query evaluation cost, since query evaluation takes at least as much time or memory as the number of results. It is therefore expected that queries with a high number of matches also score high on the absolute difficulty metric. To compensate for this, the relative difficulty metric (`relDifficulty`) is defined as $\ln \frac{\prod_c (1+|c|)}{1+\texttt{countMatches}} = \sum_c \ln(1 + |c|) - \ln(1 + \texttt{countMatches})$, expressing the logarithm of the "challenge" the query poses – this is how much worse a query engine can do than the lower bound. If the relative metric is a low figure, than the cost of query evaluation will not be much worse than the optimum, regardless of the query evaluation strategy. It can be easily shown that if a part of a graph pattern is extracted as a helper pattern that is used via pattern composition, then the sum of the relative difficulties of the two resulting patterns will be the same as the relative difficulty of the original pattern. This suggests that this metric should be treated as additive over dependent queries, and also that it is worth extracting common parts of multiple patterns into reusable helper patterns.

2.2.3 Modified Metamodel

The metamodel used for merics evaluation is depicted in Figure 2.19. This is similar to the one presented in Section 2.1.3, but cardinality constraints are mostly omitted, and *height* and *year* integer attributes are introduced for classes Segment and Sensor respectively. This gives higher freedom for the instance model generator, and more queries can be defined for attribute checks.

2.2.4 Instance Models for the Modified Metamodel

Models are characterized by their size and structure. Here size refers to the cardinalities of node and edge types. The fact that increasing model size tends to increase the cost of queries is intuitively self evident (and also empirically confirmed e.g. by our previous experiments [10, 11]). Handling large models in real life is a great chellenge, but model structure (that determines which nodes are connected to each other by which edges) must also be taken into account, which here means edge distribution of nodes. Different edge distributions also present in real-world networks: the internet or protein-protein interaction networks show *scale-free* characteristics [9], while in other areas self-healing algorithms for *binomial computer networks* are studied [8]. Average degree can impact

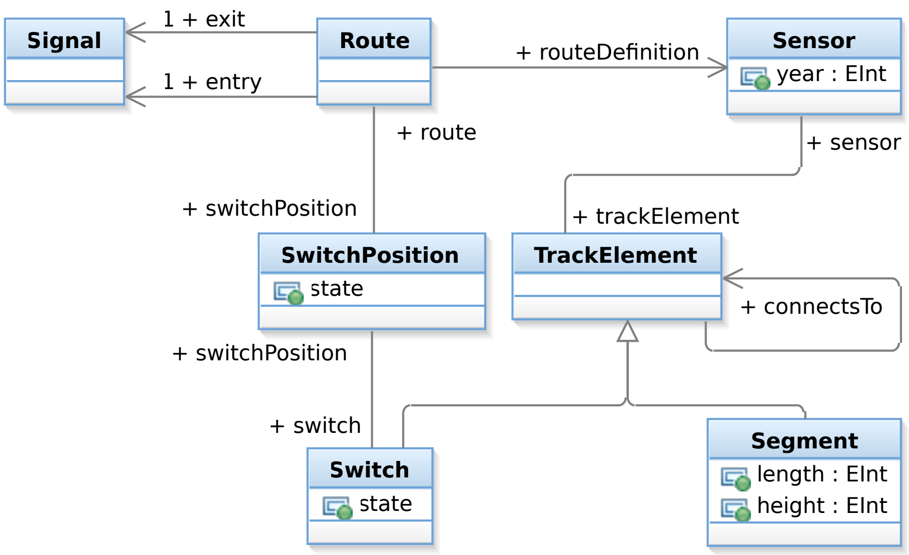


Figure 2.19: The railway metamodel (metrics version).

performance greatly, which is a typical property of different model kinds. For example, software models have usually nodes with *low degree*, while social models are usually *dense graphs*.

We have conducted the experiments on synthetic models, generated automatically with our model builder, belonging to three *model families* (without comparing them directly to real world ones, leaving it as a future work). All generated models within a family have the same approximate size; though there is some random variation due to the generation process (see later), with low standard deviation (e.g. measured as 0.3% in family A). Family A models are relatively dense graphs (~26 edges/nodes, i.e. metric `avgOutDegree`) with 1.8 million total model elements (metric `numTriples`); family B models are equally dense, but are scaled back to only 113 thousand elements; finally family C models have almost 1.3 million model elements that form a relatively sparse graph ($8.4 - 8.7$ edges/nodes).

Each model of a family has the same (expected) number of instances for any given type. However, these models of the same size still differ in their internal *structure*. Given the cardinalities of each type, our generator first created the instance sets of node types, along with generating attribute values according to an associated distribution. Then for each edge type, the generator created edge instances (with the given expected cardinality) between instances of the source type and instances of the target type. The structure of the graph is induced by the method of choosing which source node and which target node to connect. We have applied the following four methods, each taking as input the set S of source node candidates, the set T of target node candidates, and the expected cardinality e of the edge type.

Binomial case. Inspired by the well-known Erdős-Rényi model of random graphs [15], the first approach is to take each pair of source and target nodes, and draw an edge between them with a given probability p. This makes the expected cardinality of edges $e = p \times |S| \times |T|$, thus p is chosen as $\frac{e}{|S|\times|T|}$. The degrees of nodes will be binomially distributed, e.g. out-degrees with parameters $|T|$ and p.

Model Family	Model structure	countNodes	countEdges	countTriples	countTypes	avgOutDegree	avgInDegree	maxOutDegree	maxInDegree
A	Regular	63289	1646386	1811752	7	26.01	26.01	63288	44
A	Binomial	63289	1649179	1814545	7	26.06	26.06	63288	69
A	HyperGeo	63289	1646386	1811752	7	26.01	26.01	63288	74
A	Scalefree	63289	1660033	1825399	7	26.23	26.23	63288	10390
B	Regular	3954	102839	113170	7	26.01	26.01	3953	44
B	Binomial	3954	102984	113315	7	26.05	26.05	3953	64
B	HyperGeo	3954	102839	113170	7	26.01	26.01	3953	69
B	Scalefree	3954	96029	106360	7	24.29	24.29	3953	918
C	Regular	120001	1040000	1280001	7	8.67	8.67	120000	13
C	Binomial	120001	1041323	1281324	7	8.68	8.68	120000	30
C	HyperGeo	120001	1040000	1280001	7	8.67	8.67	120000	29
C	Scalefree	120001	1012858	1252859	7	8.44	8.44	120000	8929

Table 2.3: Values of model metrics on the generated instance models.

Hypergeometric case. While the previous solution ended up with a random number of edges (with expected value e), this slightly different approach will generate exactly e edges, by taking each pair of source and target nodes, and randomly selecting e from them into the graph. The degrees will be hypergeometrically distributed, e.g. out-degrees with parameters $|S| \times |T|$, $|T|$ and e.

Regular case. In software engineering models, one often finds for a given edge type that out-degrees of all nodes of the source type are roughly equal, and the same is true for in-degrees. This motivated a method that tries to uniformly (but randomly) divide e edges between the source nodes, so that the difference between any two out-degrees is at most 1; while also dividing the same edges between the target nodes, with a similar restriction on in-degrees.

Scale-free case. It has been observed in many different disciplines that degree distributions of certain large graphs follow a power law, especially growing / evolving graphs with the *preferential attachment* property (a new edge is more likely to connect to a node which already has a higher degree). We have used a variant of the preferential attachment bipartite graph generator algorithm of [31] to generate the connections from source nodes to target nodes.

The four generation methods induced significantly different degree distributions. This and other differences are shown in Table 2.3.

One-to-many relationships were treated in a special way to meet the multiplicity restriction. In particular, a single top-level container element (not depicted in the metamodel figure, neither involved in any queries) was used to contain all elements; it therefore has an outgoing containment edge for every other object, thereby "polluting" the `maxOutDegree` metric.

2.2.5 Benchmark Queries

Based on the previously defined metrics for query specification and based on the metamodel, six series of model query specifications were systematically constructed. Each query series includes four to seven model queries that aim to be different in only one of the defined model query metrics. Executing a query series on the same model and tool results in a data series that shows how the tool is scalable according to the represented model metric.

`Locals` Query Series for the `numVariables` Metric

Five queries were defined, where each one includes the same number of edge constraints, but the number of local variables increases.

It can be realized using the `Segment` type and `connectsTo` reference, so it means that only one node type and reference type is used in these patterns and the focus is on the structure of the patterns. The simple graph based visualisation of these pattern structures is shown in Figure 2.20, where the node drawn with empty circle represents the single pattern parameter.

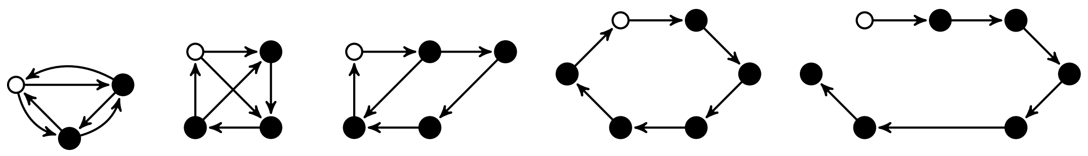


Figure 2.20: The `Locals` patterns.

`Refs` Query Series for the `numEdgeConstraints` Metric

The `Refs` query series is also constructed based on the `Segment` type and `connectsTo` reference.

Here, the number of edge constraints increases along the series, but the number of local variables is constant in all of the generated four queries. The visualisation of these pattern structures is shown in Figure 2.21.

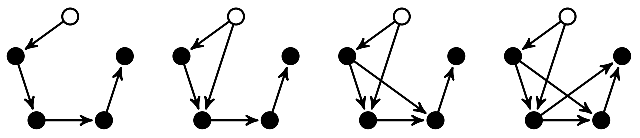


Figure 2.21: The `Refs` patterns.

`Params` and `ParamCircle` Query Series for the `numParameters` Metric

Two series of queries were constructed for this metric, because of performance reasons. The `Params` query series is the more complex and some tools exceed the time limit in the benchmark. The `ParamsCircle` query series is a simplification of the `Params` series.

The goal of this constructed query series is to create patterns with the same body, but with an increasing number of parameters. The first of these queries (returning one parameter) is shown in Figure 2.22, where the parameter is the blue object. Other queries use the same body, but add sen1, then sen2, then other variables to the parameter list.

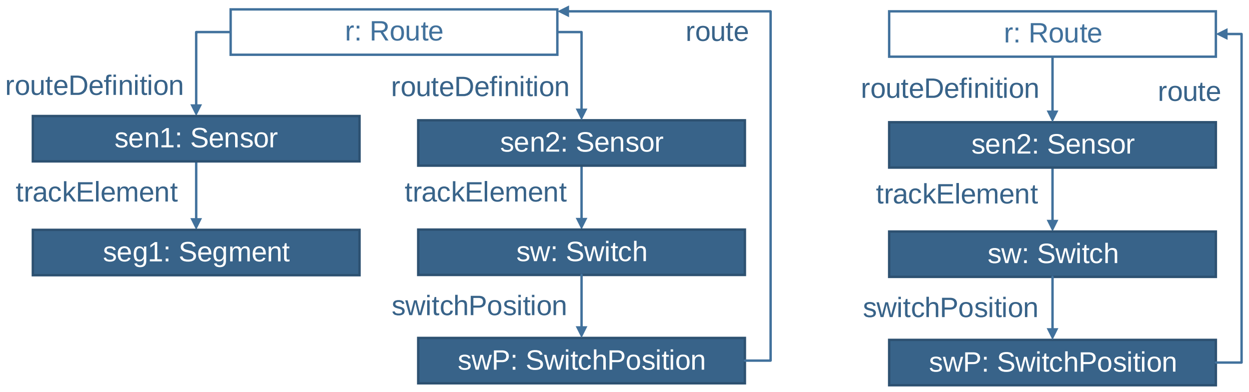


Figure 2.22: The `Params` and `ParamsCircle` pattern schemas (first step).

`Checks` Query Series for the `numAttrChecks` Metric

Each `Checks` query use the same pattern body described by the pattern schema in Figure 2.23, but each one is extended with an increasing number of attribute check constraints. These check constraints filter results based on the *year, height* and *length* value of the segments *seg1* and *seg2*, resulting in seven queries.

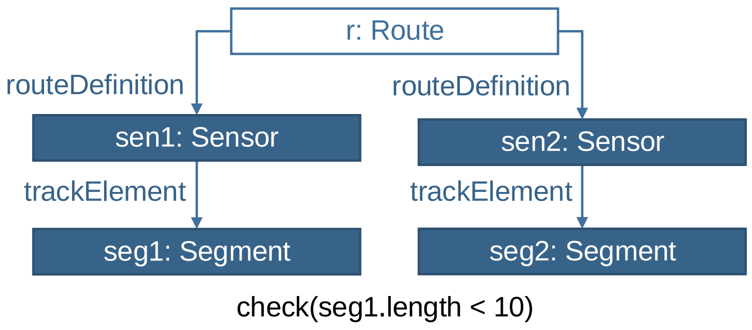


Figure 2.23: The `Checks` pattern schema.

`Negs` Query Series for the `nestedNacDepth` Metric

`Negs` queries present increasing number of nested *neg* constraints. These queries are defined based on the following schema: at the bottom there is a pattern checking for segments with length less than ten. Next, for each query a new segment is matched, and the previous pattern is encapsulated in a negative pattern call. $i = 5$ queries are defined in the benchmark, described by the schema in Figure 2.24.

After the construction of the query series, we evaluated the query only metrics on them. Table 2.4 shows the results of the evaluation: each cell contains the value or range of values that we got on

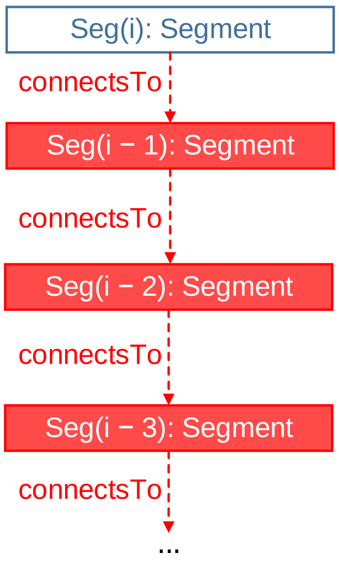


Figure 2.24: The `Negs` pattern schema.

Query series	numParameters	numVariables	numEdgeConstraints	numAttrChecks	nestedNacDepth
Param	1–5	8	8	0	0
ParamCircle	1–5	6	6	0	0
Locals	1	3–7	6	0	0
Refs	1	5	4–7	0	0
Checks	1	5–11	4–10	0–6	0
Neg	2–6	3–11	1–5	1	0–10

Table 2.4: Query-only metrics.

each query series. This table confirms that there are query series for every metric (shown in blue), and each query series differ in one or more metrics.

2.2.6 Complex Analysis

2.3 Implementation architecture

In this chapter, we discuss the implementation details of the Train Benchmark. The installation guide is available in the Train Benchmark core repository.[3]

For the integration of the Train Benchmark projects and their third party dependencies, we use Apache Maven [1]. The dependencies are shown in Figure 2.25.

In this section, we briefly describe the tasks and responsibilities of each module. The modules are grouped to Maven *build profiles* which can be built separately.

[3]`https://github.com/FTSRG/trainbenchmark-core`

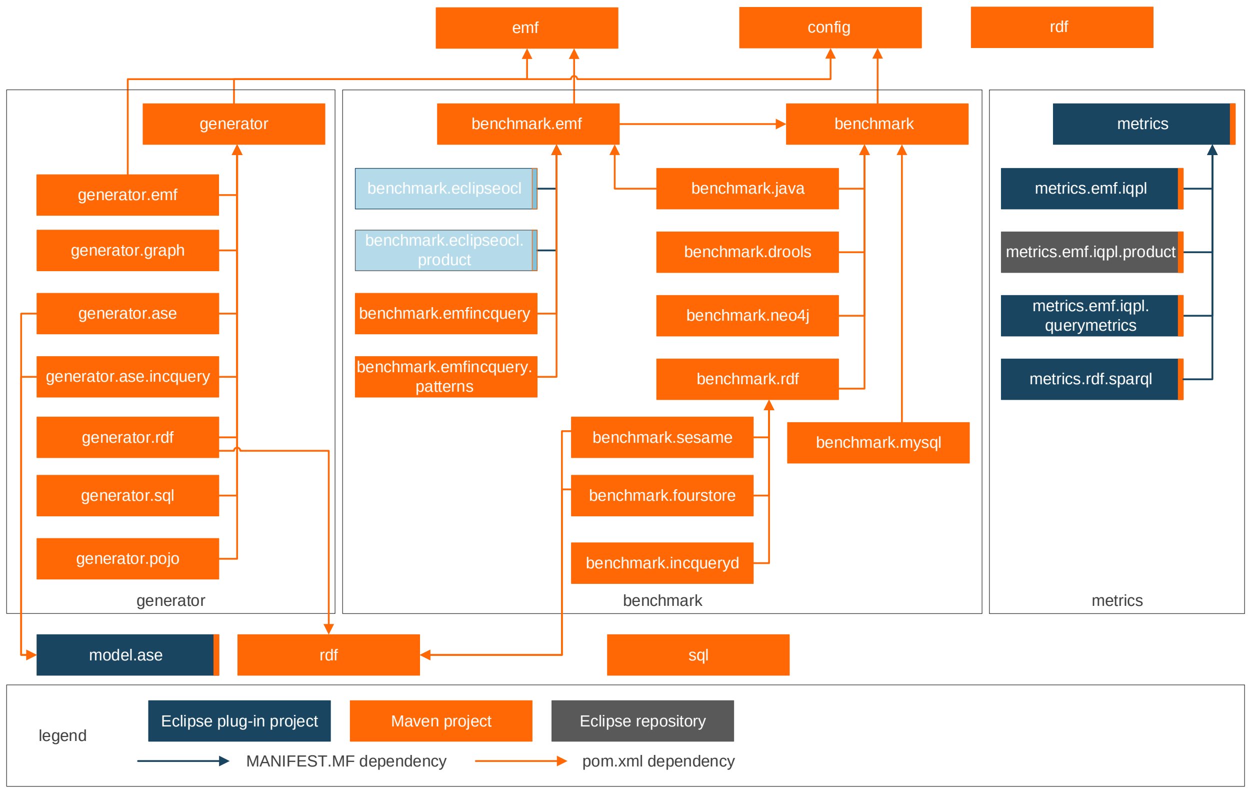


Figure 2.25: The Maven modules defined in the Train Benchmark. Note that artifact ids of the modules are shortened and the full ids start with `hu.bme.mit.trainbenchmark`.

2.3.1 Parent module

The `hu.bme.mit.trainbenchmark` module is the parent module which contains the modules used in the Train Benchmark. Building this Maven module builds all child modules as well.

2.3.2 Central modules

The `hu.bme.mit.trainbenchmark.model` module contains the reference metamodel represented in EMF.

The `hu.bme.mit.trainbenchmark.config` module contains classes and constants used by the `generator` and the `benchmark` projects.

2.3.3 Representation-specific modules

The `hu.bme.mit.trainbenchmark.emf` and `hu.bme.mit.trainbenchmark.rdf` modules contain classes and constants used by the particular representations.

2.3.4 Generator modules

The `hu.bme.mit.trainbenchmark.generator.*` modules are responsible for generating the instance models for the benchmarks.

- `emf`: generates an EMF instance model.
- `emfuuid`: generates an EMF instance model with UUIDs. A UUID (universally unique identifier) identifies an EObject uniquely, and is generated automatically by the EMF framework. In the output file, UUIDs are used to reference other objects, and not XPath expressions, speeding up the serialization process.
- `graph`: generates a property graph model in the specified format: GraphML (default), Blueprints GraphSON, Faunus GraphSON.
- `rdf`: generates an RDF instance model.
- `sql`: generates an SQL script which creates and loads the appropriate database tables.

2.3.5 Benchmark modules

The `hu.bme.mit.trainbenchmark.benchmark.*` modules are responsible for benchmarking. For the list of current implementations, see Table 3.1.

2.3.6 4store

To access 4store through a graph-like API, we developed a Java client with a focus on high performance[4]

[4] `https://github.com/FTSRG/rdf-graph-drivers`

Chapter 3

Benchmark Results

3.1 Benchmarking Environment

For the implementation details, source codes and raw results, see the benchmark website[1]. In this section we describe the runtime environment, and highlight some design decisions.

The benchmark machine contains two dual-core Intel Xeon (3.00 GHz) CPU, 12 GBs of RAM and an SAS disk formatted to ext4 for storing the models. In order to alleviate disturbance of a running measurement and minimize noise in the results, a bare metal 64-bit Ubuntu 12.10 OS was installed with unnecessary services (like `cron`) turned off. Oracle JVM version 1.7.0_51 is used as the Java environment and Eclipse Kepler Modeling 64-bit for Linux to satisfy specific tool dependencies.

The performance measurements of a tool was independent from the others, i.e. for every tool only its codebase was loaded, and every measurement of a scenario was started in a different JVM. Before the execution, the OS file cache was cleared, and swapping was disabled to avoid this kind of thrashing. Each test case (including all phases) must be run within a specified time limit (15 minutes), otherwise its process was killed.

In the benchmark all cases were run 10 times, and the results were dumped into files, which were aggregated using the R statistical framework. The correlation results and performance plots are written into an HTML report.

3.2 Tools

The measured tools generally work on graph-based models (like EMF [28] or RDF [32]), and provide a graph pattern-like query language. Table 3.1 shows the list of currently integrated tools.

Note on **incrementality**: *incremental* means that the tool not only employs caching techniques, but provides a dedicated incremental query evaluation algorithm that processes *changes* in the model and propagates these changes to query evaluation results in an incremental way (i.e. avoiding complete recalculations). Both EMF-INCQUERY and JBoss Drools are based on the Rete algorithm; OCL also

[1]`https://incquery.net/publications/trainbenchmark/full-results`

Tool	Version	Model DL	Query language	Incremental	In-memory only	Implementation language
Java	7.0	EMF	Java	❍	●	C++
Eclipse OCL	3.3.0	EMF	OCL	❍	●	Java
EMF-INCQUERY	0.7.2	EMF	IQPL	●	●	Java
Drools	5.4.0	EMF	DRL	●	●	Java
Sesame	2.7.9	RDF	SPARQL	❍	●	Java
4store	1.1.5	RDF	SPARQL	❍	❍	C
Neo4j	1.9.2	Graph	Cypher	❍	❍	Java

Table 3.1: Tools used in the benchmark.

has an incremental extension called the *OCL Impact Analyzer* which is planned to be included in the measurements in the final version of the tech report.

In contrast, *non-incremental* tools are using *search-based* algorithms that evaluate each query with a model traversal, which may be optimized using heuristics and/or caching mechanisms. The essential difference to incremental tools is that they do not process changes in the model, but only take the current state of the model as the input.

Note on **in memory operations**: most of the integrated tools use models that are materialized into operating memory (RAM), hence query evaluation does not involve (expensive) disk operations. Some tools do not support in-memory backends, so to compensate for the disadvantage, such tools are used with files that are stored on RAM disks.

3.2.1 EMF-based Tools

Java

An imperative *local search-based* approach was implemented in Java, operating on *Eclipse Modeling Framework (EMF)* [28] models. Queries are implemented as Java functions, traversing a model without any search plan optimization, but they cut unnecessary search branches at the earliest possibility.

Eclipse OCL

The OCL [22] language is commonly used for querying EMF model instances in validation frameworks. It is a standardized navigation-based query language, applicable over a range of modeling formalisms. Taking advantage of the expressive features and wide-spread adoption of this query language, the project Eclipse OCL [14] provides a powerful query interface that evaluates such expressions over EMF models.

EMF-INCQUERY

EMF-INCQUERY [11] is an Eclipse Modeling project that provides incremental query evaluation using Rete [16] nets. Queries can be written in its graph pattern based query language (IncQuery Pattern Language, IQPL [12]), which is evaluated over EMF models.

Drools

Incremental query evaluation is also supported by the *Drools* [4] rule engine developed by Red Hat. It is based on a variant of Rete [16] (object-oriented Rete). Queries can be formalized using its own rule description language. Queries can be constructed by naming the ”when” part of rules and acquiring their matches. While Drools is not an EMF tool per se, the Drools implementation of the Train Benchmark works on EMF models.

3.2.2 RDF-based Tools

Sesame

Sesame gives an API specification for many tools, and also provides its own implementation. The tool evaluates queries over RDF that are formulated as SPARQL [33] graph patterns.

4store

4store [19] is an open source, distributed triplestore implemented in C. The main goal of 4store is to provide a high performance storage and query engine for semantic web applications.

INCQUERY-D

INCQUERY-D [20] is a distributed incremental graph query engine developed in the Budapest University of Technology and Economics. The goal of INCQUERY-D is to provide a scalable query engine by using a cluster of servers instead of a single workstation.

3.2.3 Graph-based Tools

Neo4j

As part of the NoSQL movement, database management systems emerged with a focus on graph storage and processing. As of 2014, the most popular graph database is Neo4j [3]. The data model is based on graphs, where any node or edge can be labeled. Cypher can be used to query labeled graphs using its own graph pattern notation. This engine also uses disk for data storing, so a RAM disk is created during the benchmark.

3.2.4 SQL-based Tools

MySQL

MySQL [2] is one of the most well-known and widely used open-source relational database management systems.

3.3 Measurement Results for Performance Comparison

The measurement results of the benchmark are displayed below. These diagrams show the batch query performance and incremental evaluation time of each tool, for different model sizes. Additionally, the initial and the updated result set size is displayed under the model sizes for the batch and incremental queries, respectively.

3.3.1 How to read the charts

The charts are presented with logarithmic scales on both axes. This means that the general "linear" appearance of all measurement series correspond to a low-order polynomial big-O characteristic where the slope of the plot determines the dominant order (exponent). Moreover, a constant difference on these plots corresponds to a constant order-of-magnitude (i.e. constant multiplier) difference. As the execution runs with a timeout, incomplete measurement series indicate that the time limit has been reached during execution.

The execution phases as described in Section 2.1.1 have been grouped into several aggregated results as follows:

- *Batch validation* (Section 3.3.2) corresponds to the sum of *read* and *check* execution times, as it models a batch validation scenario where the model is read and validated.
- *Revalidation* (Section 3.3.3) corresponds to the sum of *edit-recheck* times as it represents the cumulative execution time of model editing-revalidation cycles.
- *Total time* (Section 3.3.4) represents the combined execution time of the entire measurement cycle, giving an overall performance indicator.
- *Series of edit times* (Section 3.3.5) presents a deeper analysis to show how long each *edit-recheck* cycle takes. These plots are useful for observing transient effects such as JIT HotSpot compilation.
- *Series of revalidation times* (Section 3.3.6) present the same for the *re-check* phase.

Each plot shows execution times corresponding to a combination of one of the queries of Section 2.1.4 and a tranformation scenario as defined in Section 2.1.5. The plots offer insights into performance characteristics in three major ways:

- *Execution time vs. instance model size.* Each plot can be directly interpreted as an evaluation of performance against increasing model size, where different tool characteristics can be compared in a single plot.

- *Characteristics vs. increasing query complexity.* As the queries are of different complexity, comparing the plots within the same transformation scenario group but against different queries is useful to judge tool characteristics against increasing query complexity (or query result size, shown as "Results" on the Y axis).
- *Characteristics vs. increasing transformation complexity.* As the transformation scenarios are characteristically different (e.g. with respect to the amount of objects modified by the transformation sequence – as shown on the X axis as "Modifications"), comparing the plots with the same query but against different scenarions is useful to judge tool characteristics against various transformation complexities.

3.3.2 Batch Validation

Batch Validation (Inject Scenario)

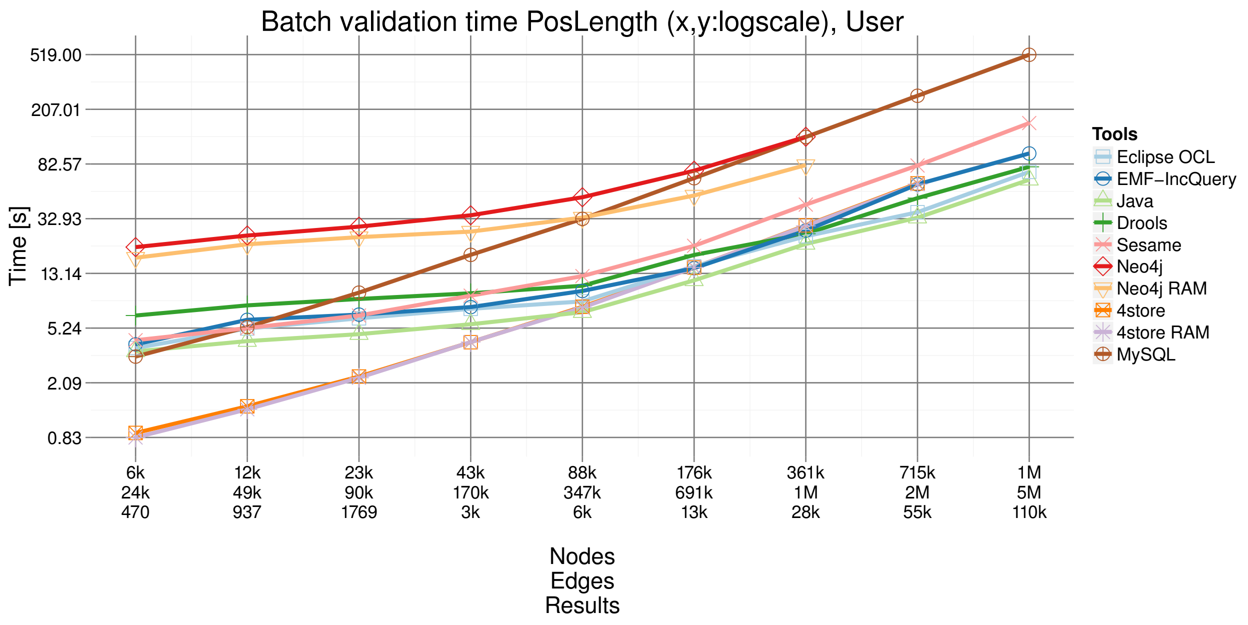


Figure 3.1: Batch validation times for the PosLength query in the Inject scenario.

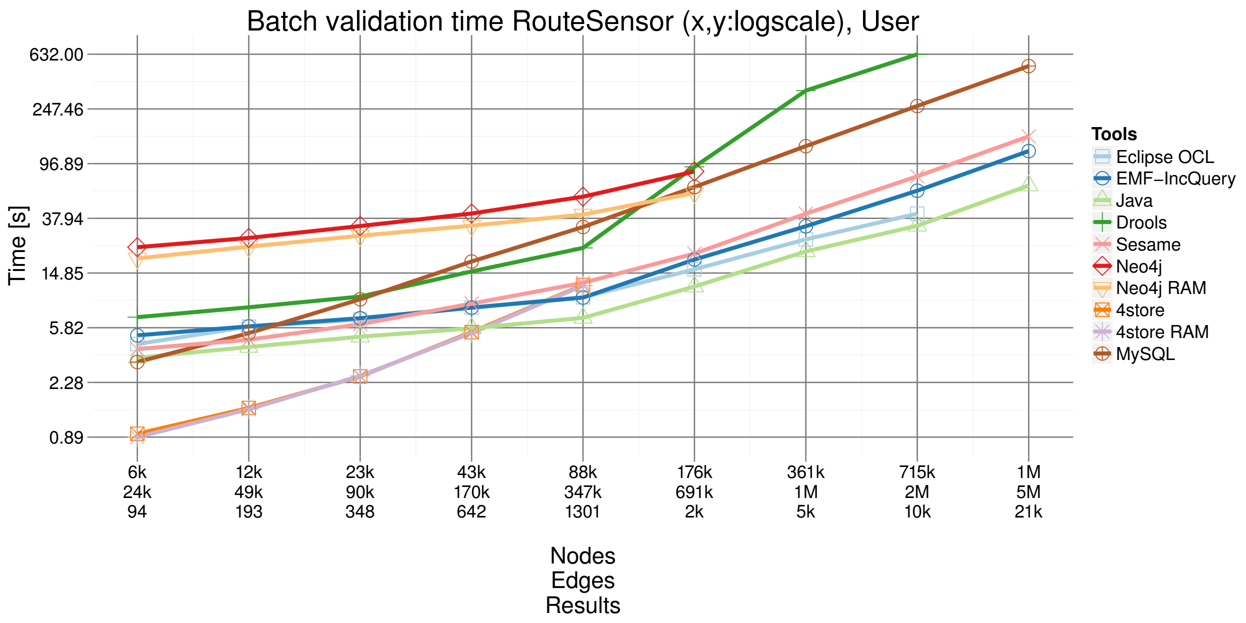


Figure 3.2: Batch validation times for the RouteSensor query in the Inject scenario.

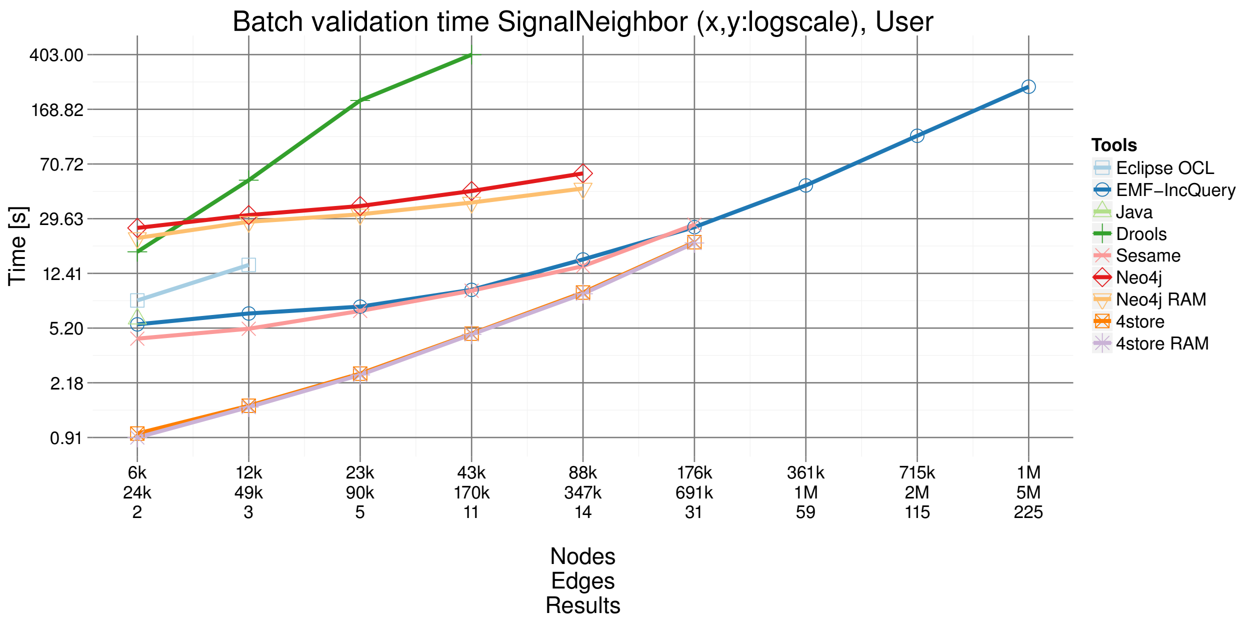


Figure 3.3: Batch validation times for the SemaphoreNeighbor query in the Inject scenario.

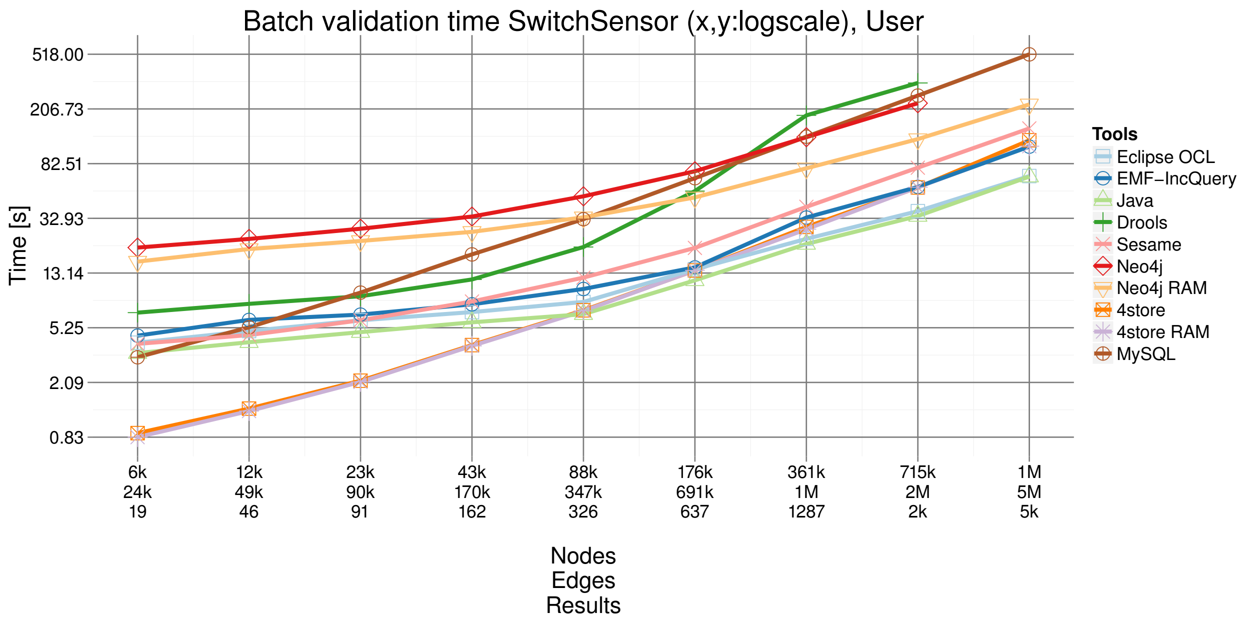


Figure 3.4: Batch validation times for the SwitchSensor query in the Inject scenario.

Batch Validation (Repair Scenario)

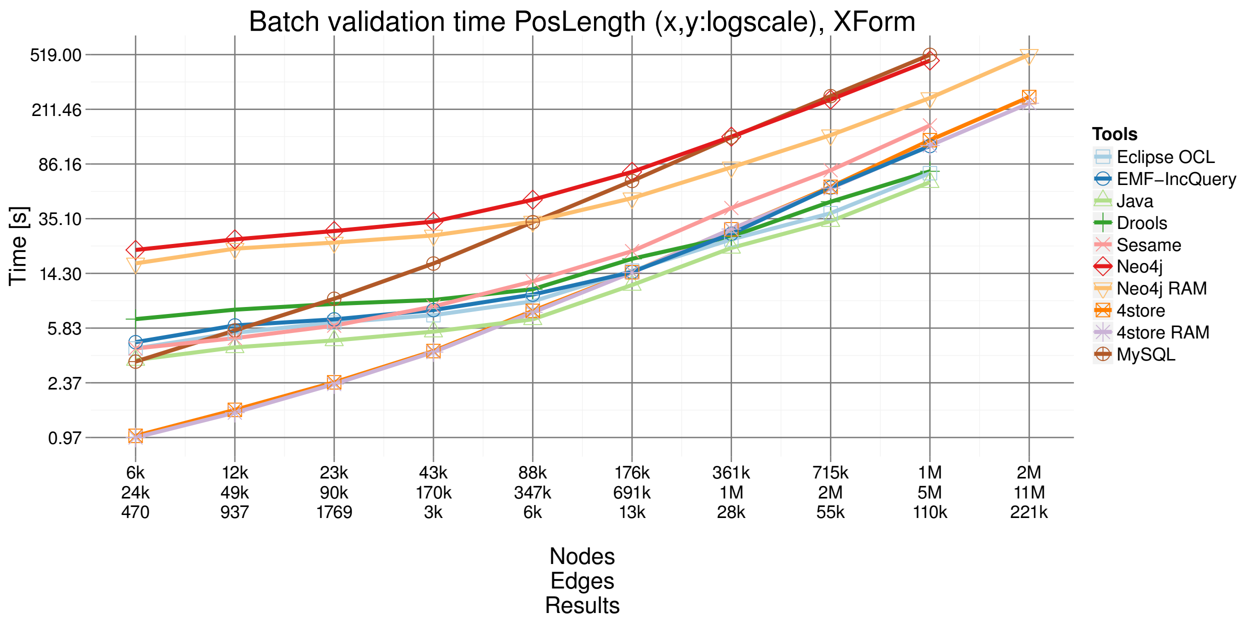


Figure 3.5: Batch validation times for the PosLength query in the Repair scenario.

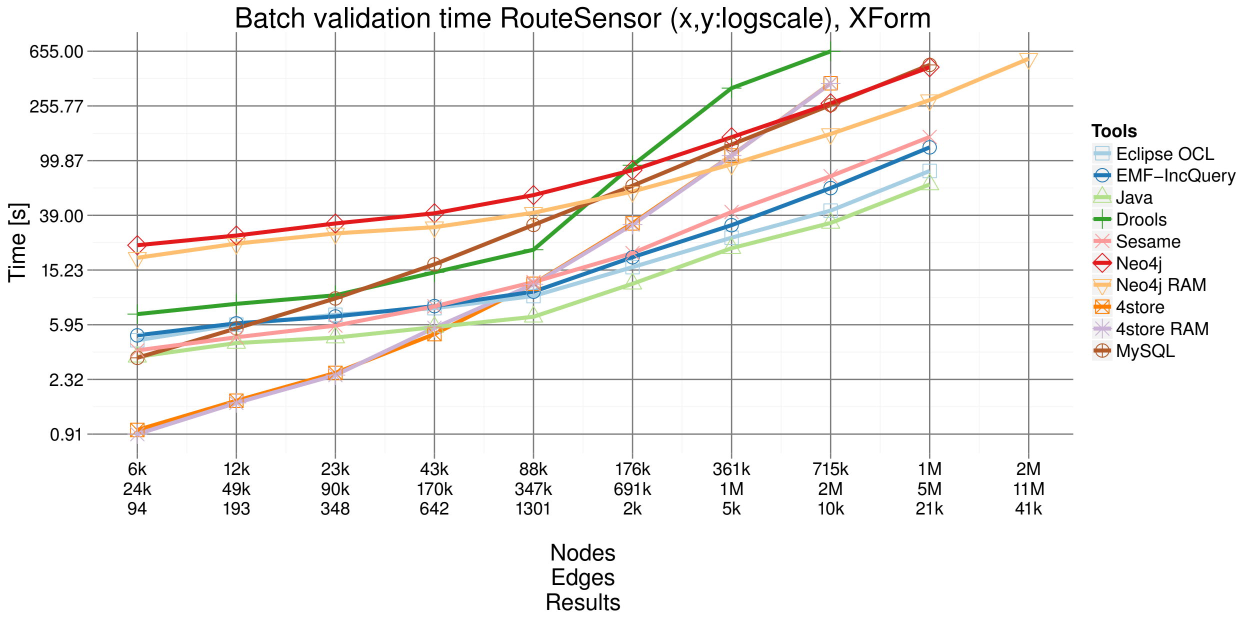


Figure 3.6: Batch validation times for the RouteSensor query in the Repair scenario.

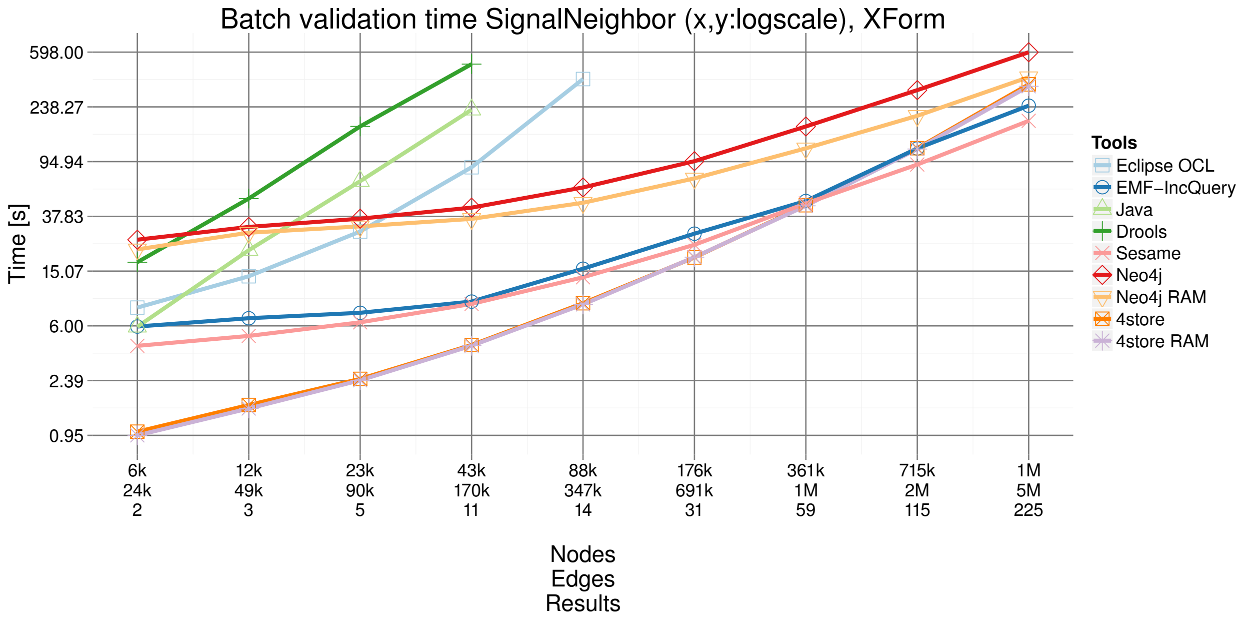


Figure 3.7: Batch validation times for the SemaphoreNeighbor query in the Repair scenario.

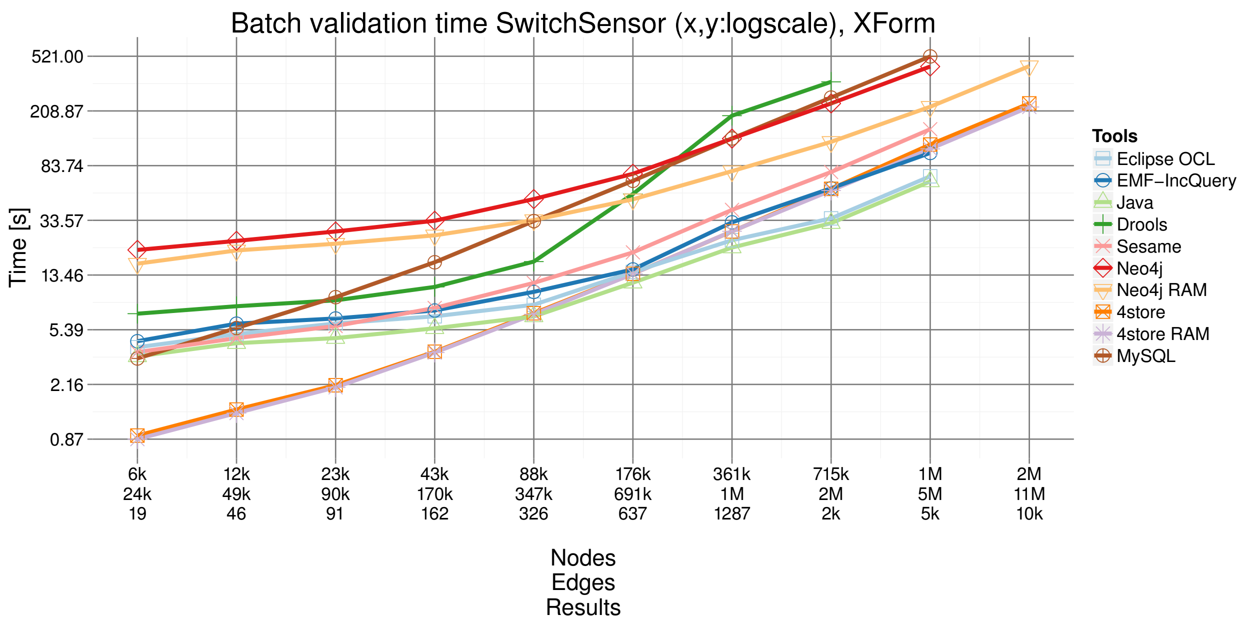


Figure 3.8: Batch validation times for the SwitchSensor query in the Repair scenario.

3.3.3 Revalidation

Revalidation (Inject Scenario)

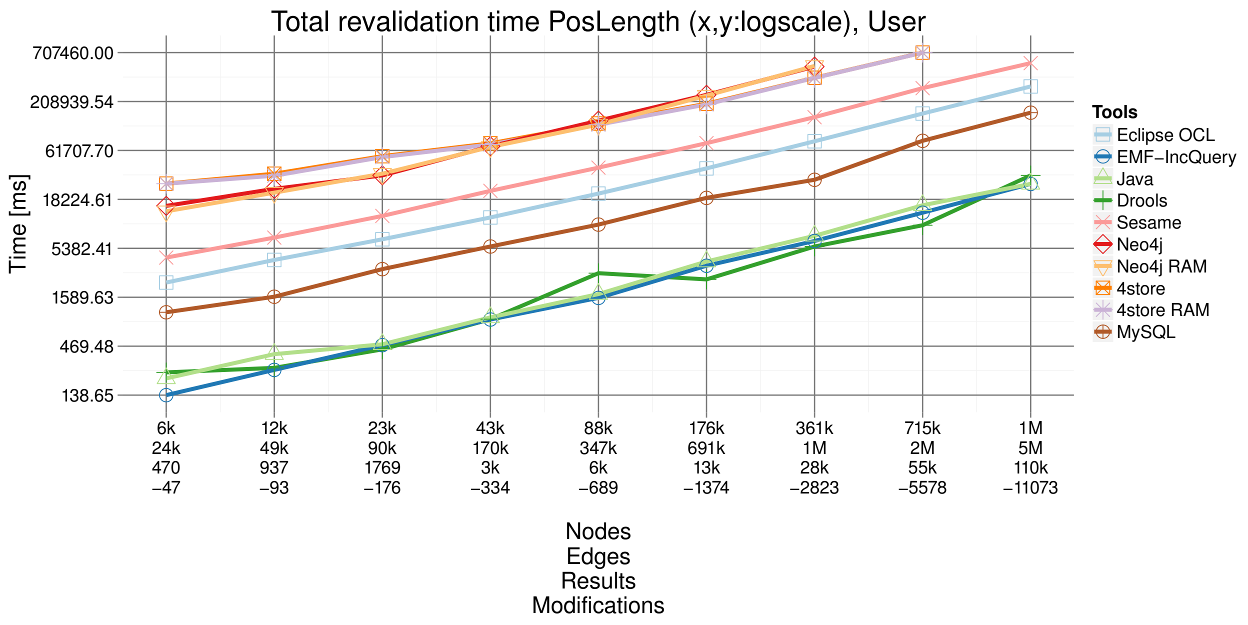


Figure 3.9: Revalidation times for the PosLength query in the INJECT scenario.

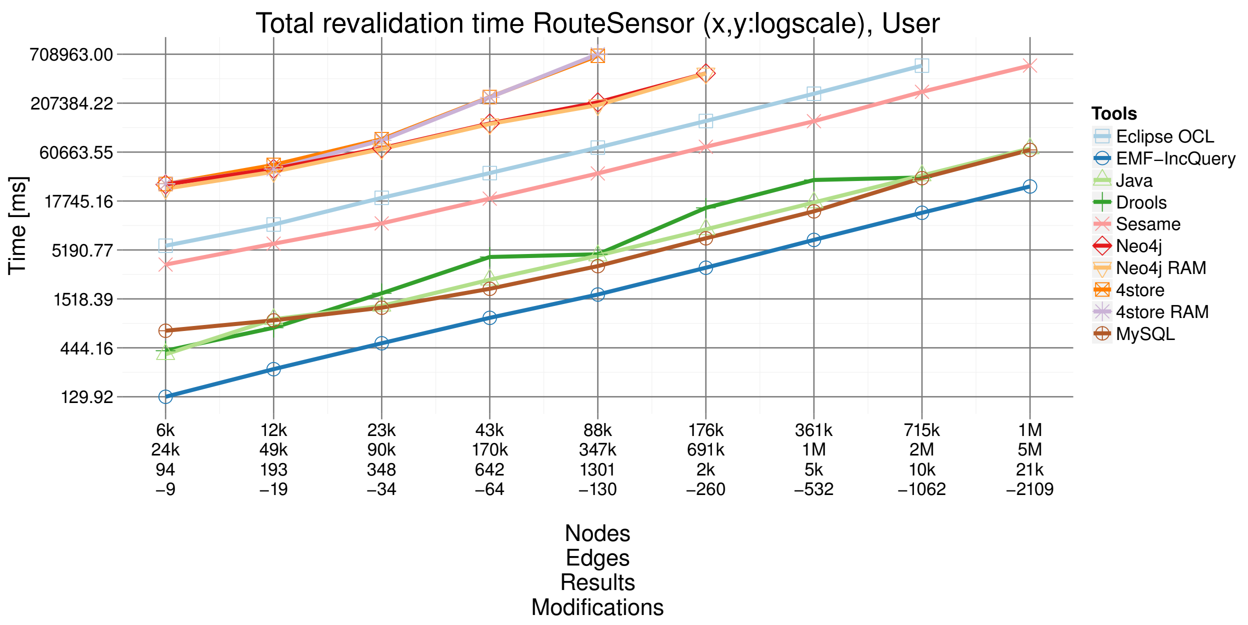


Figure 3.10: Revalidation times for the RouteSensor query in the INJECT scenario.

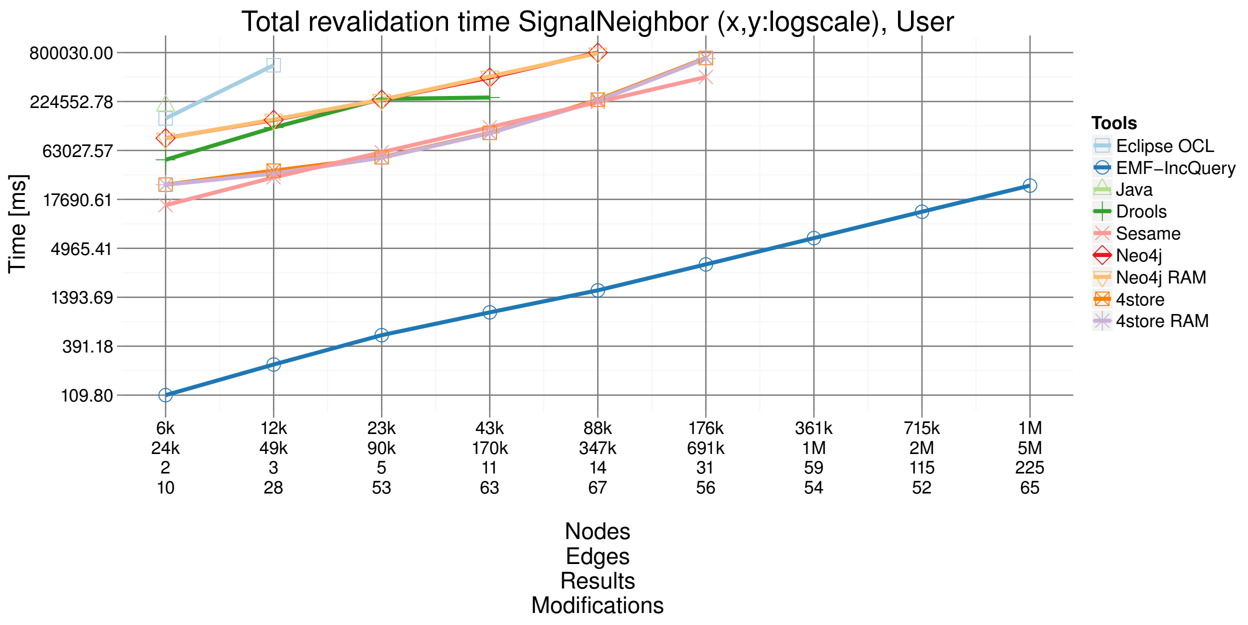


Figure 3.11: Revalidation times for the SemaphoreNeighbor query in the INJECT scenario.

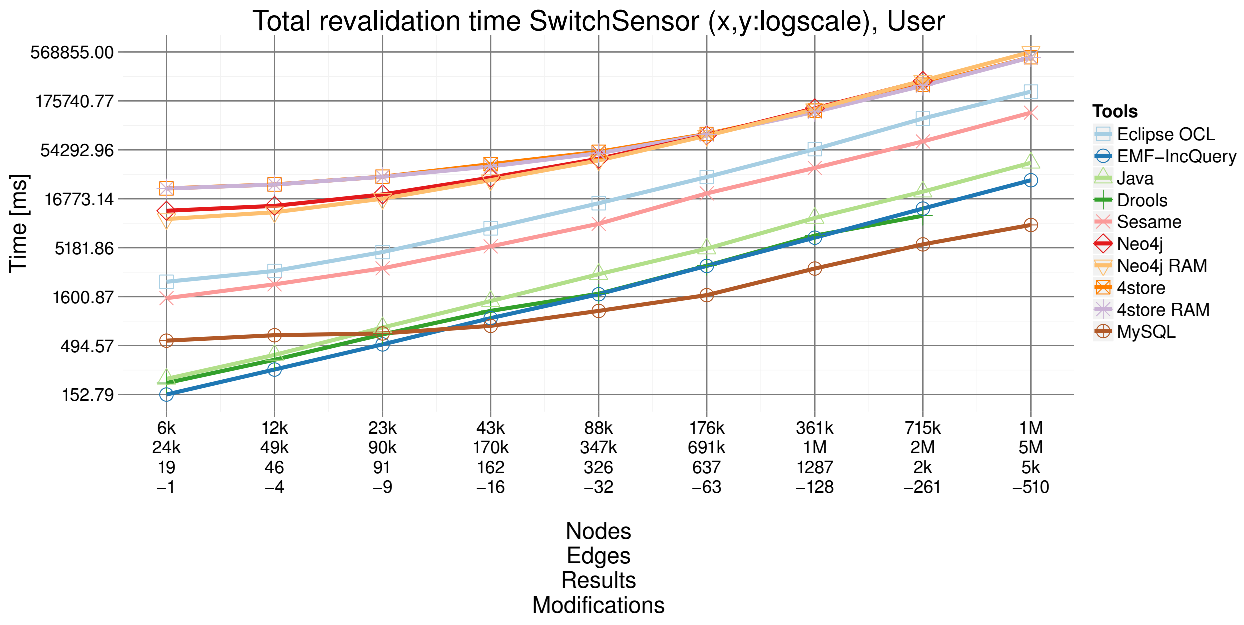


Figure 3.12: Revalidation times for the SwitchSensor query in the INJECT scenario.

Revalidation (Repair Scenario)

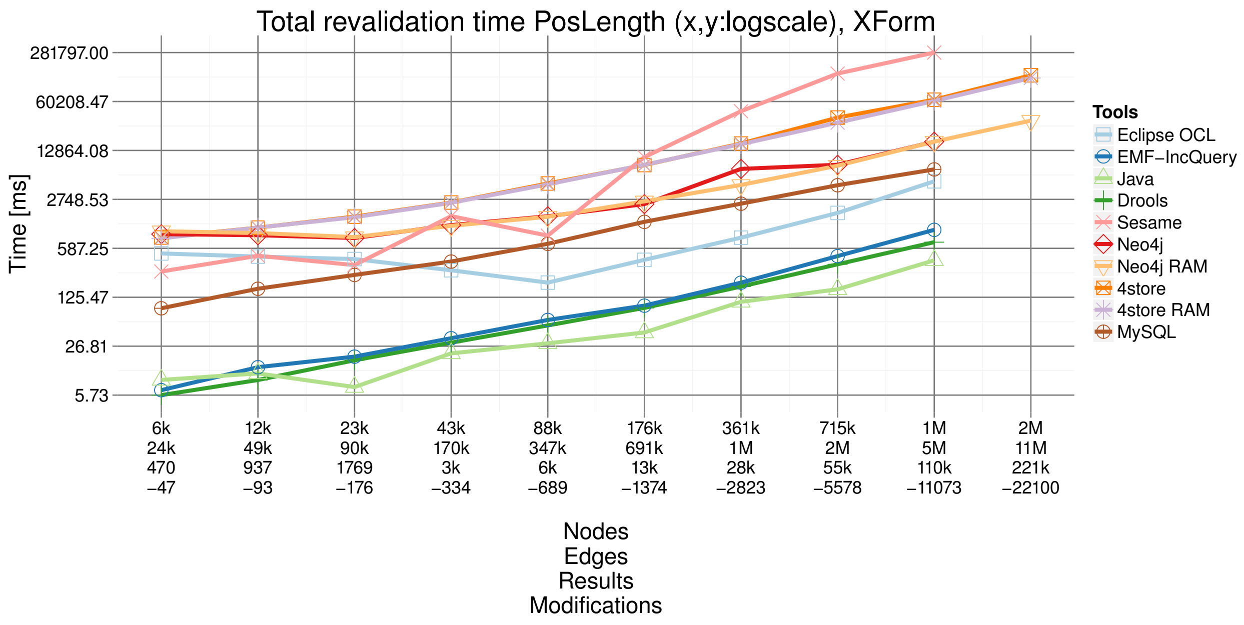


Figure 3.13: Revalidation times for the PosLength query in the Repair scenario.

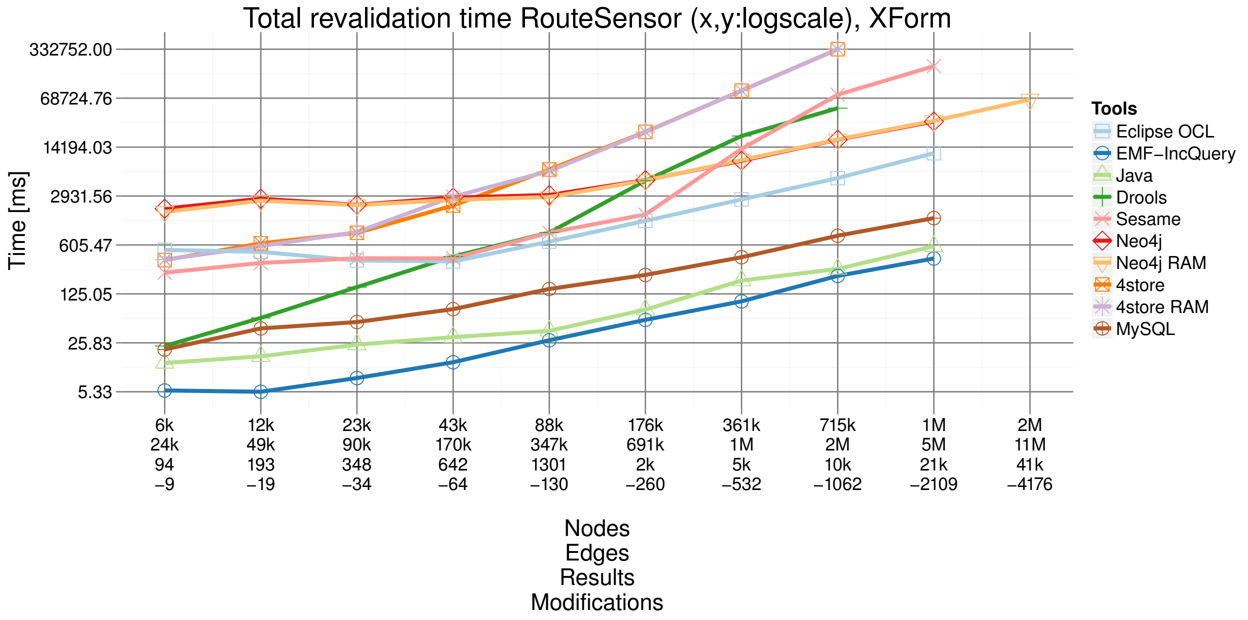


Figure 3.14: Revalidation times for the RouteSensor query in the Repair scenario.

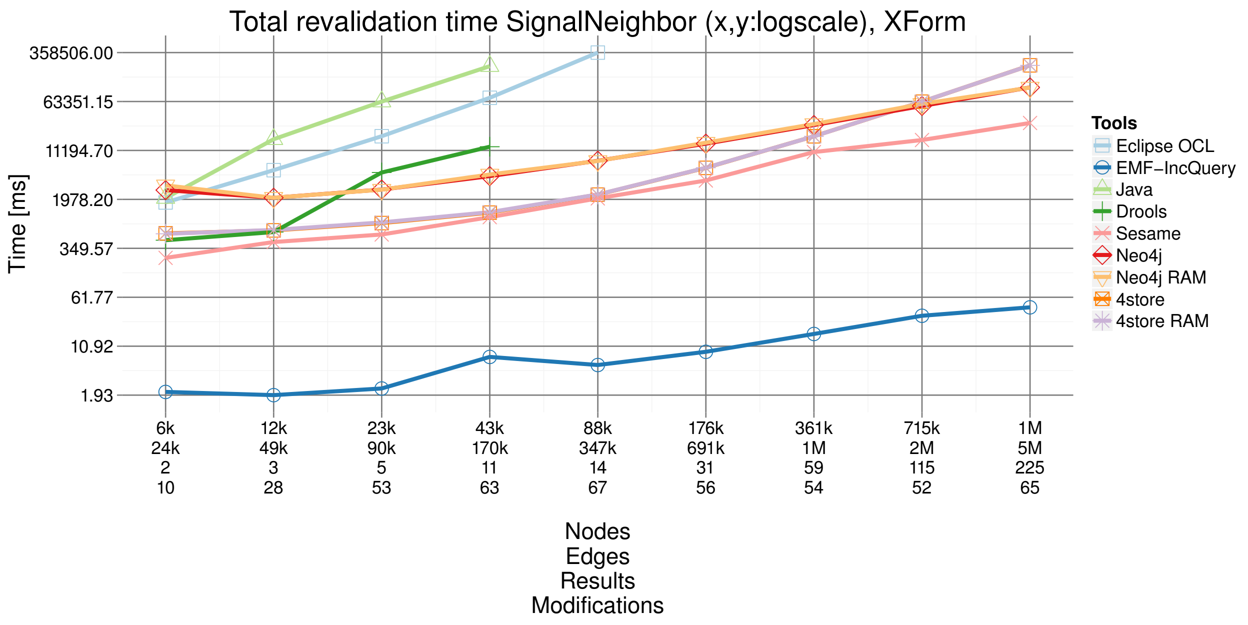


Figure 3.15: Revalidation times for the SemaphoreNeighbor query in the Repair scenario.

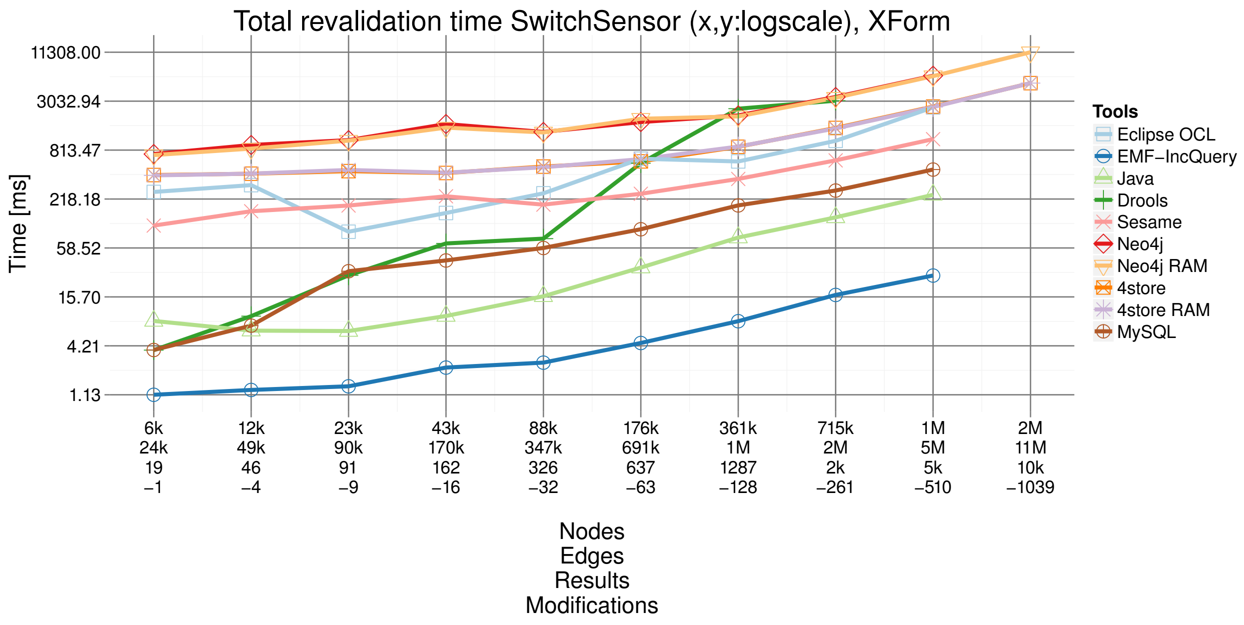


Figure 3.16: Revalidation times for the SwitchSensor query in the Repair scenario.

3.3.4 Total time

Total Time (Inject Scenario)

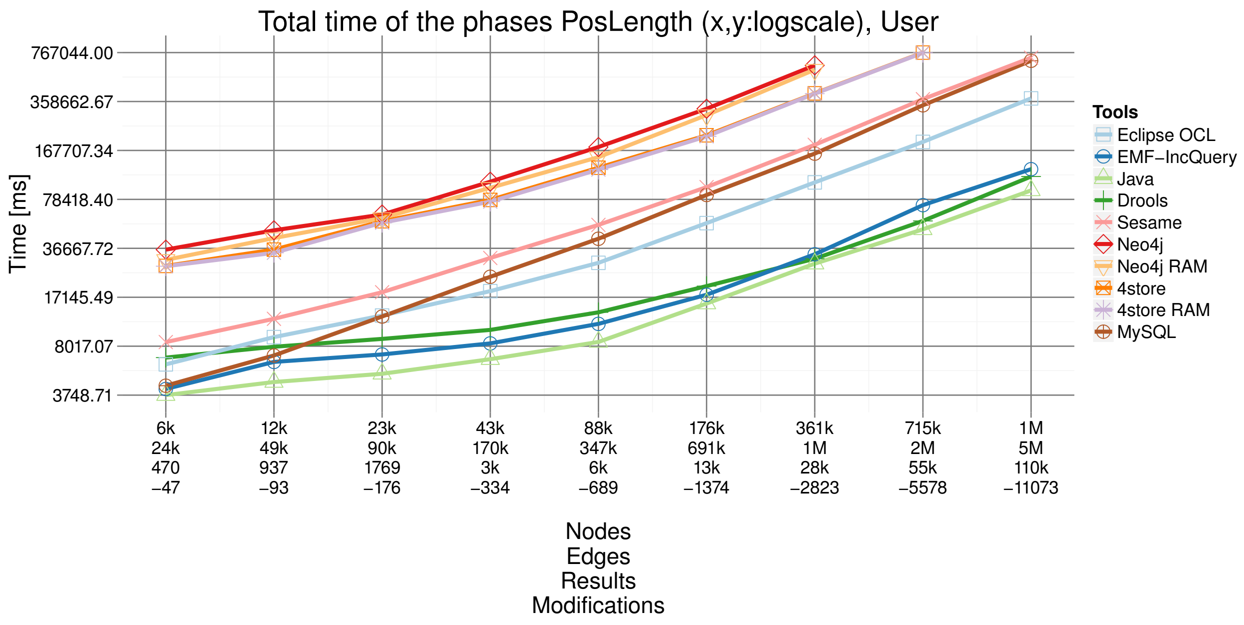


Figure 3.17: Total time for the PosLength query in the Inject scenario.

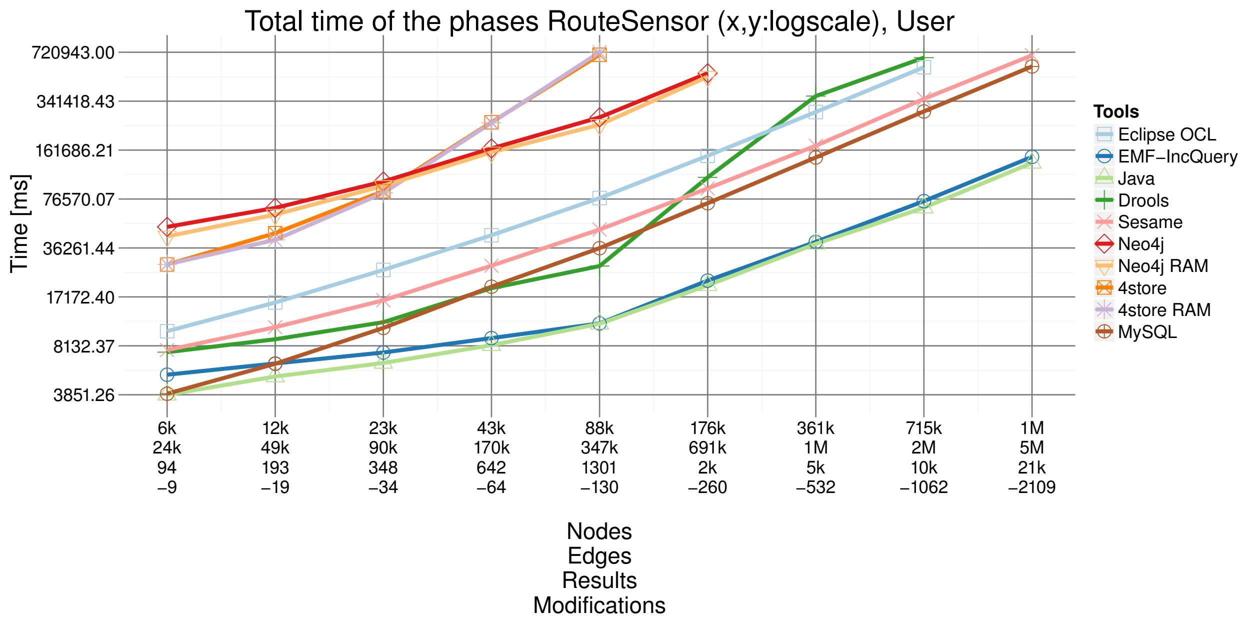


Figure 3.18: Total time for the RouteSensor query in the Inject scenario.

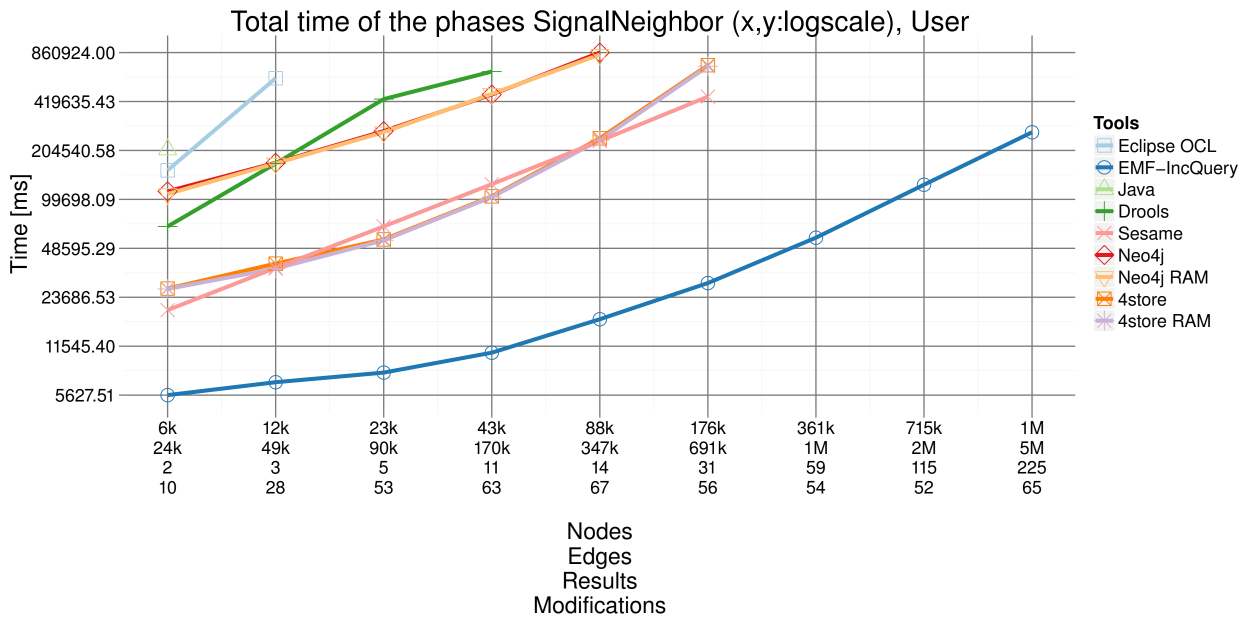


Figure 3.19: Total time for the SemaphoreNeighbor query in the Inject scenario.

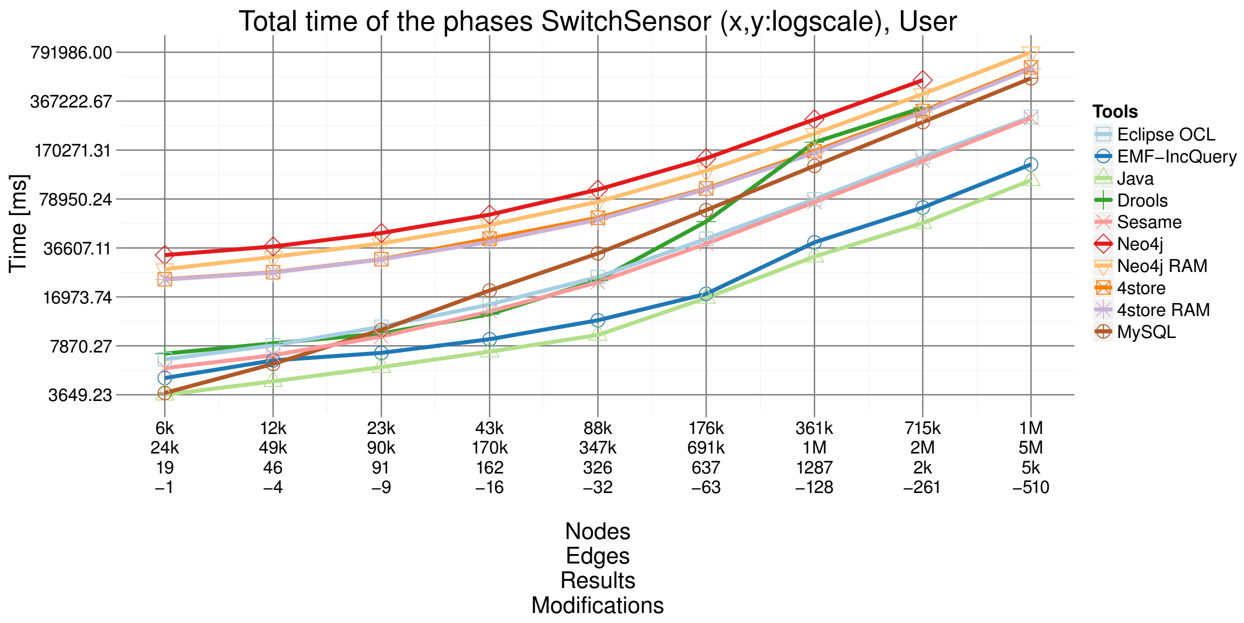


Figure 3.20: Total time for the SwitchSensor query in the Inject scenario.

Total Time (Repair Scenario)

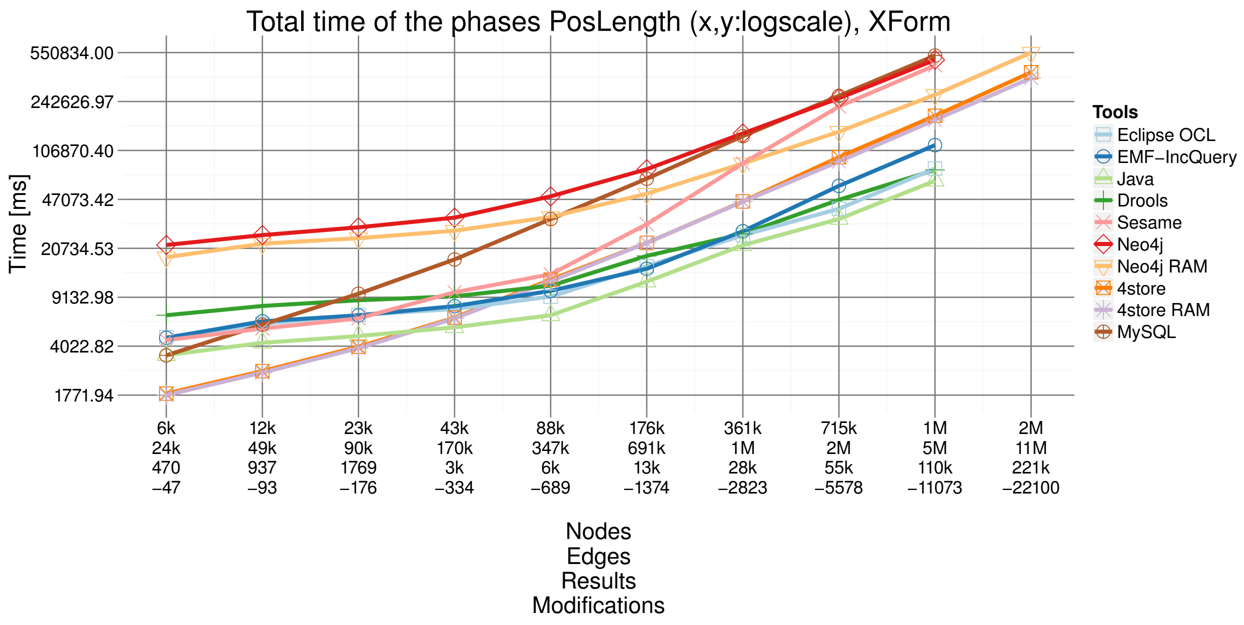


Figure 3.21: Total time for the PosLength query in the Repair scenario.

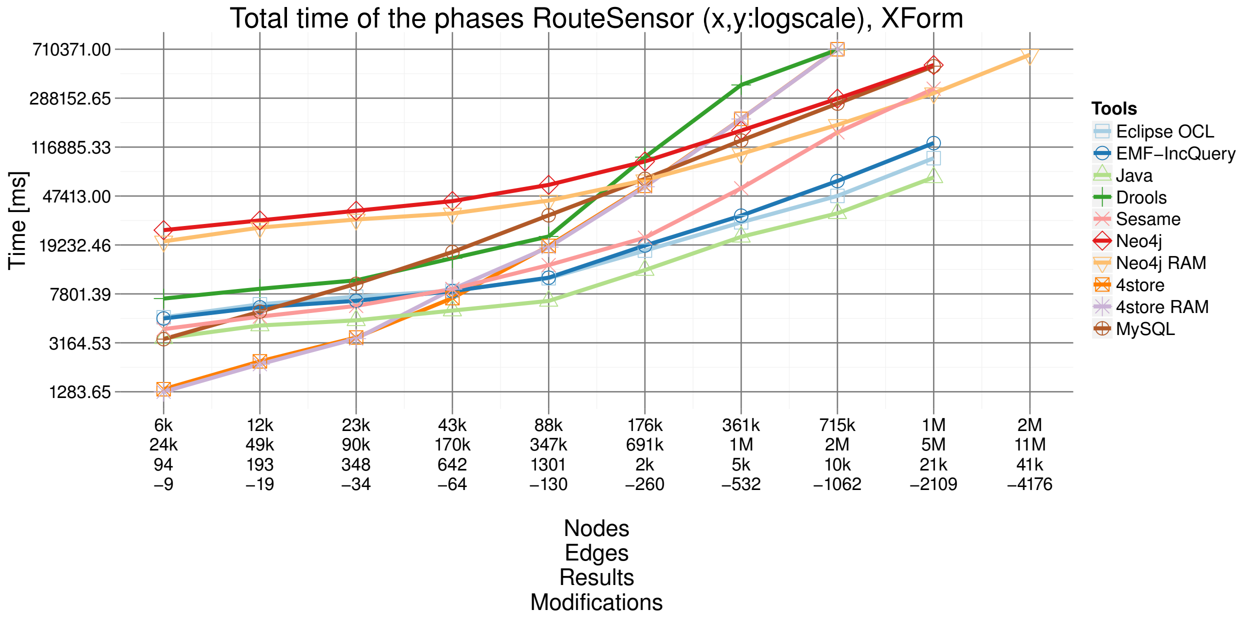


Figure 3.22: Total time for the RouteSensor query in the Repair scenario.

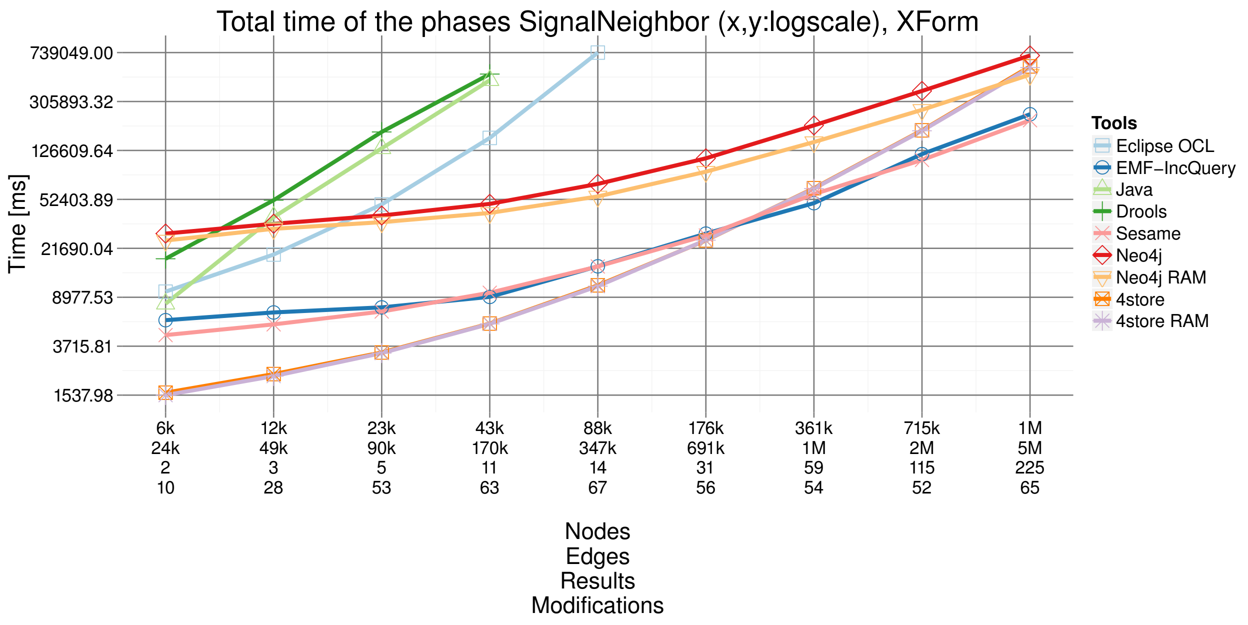


Figure 3.23: Total time for the SemaphoreNeighbor query in the Repair scenario.

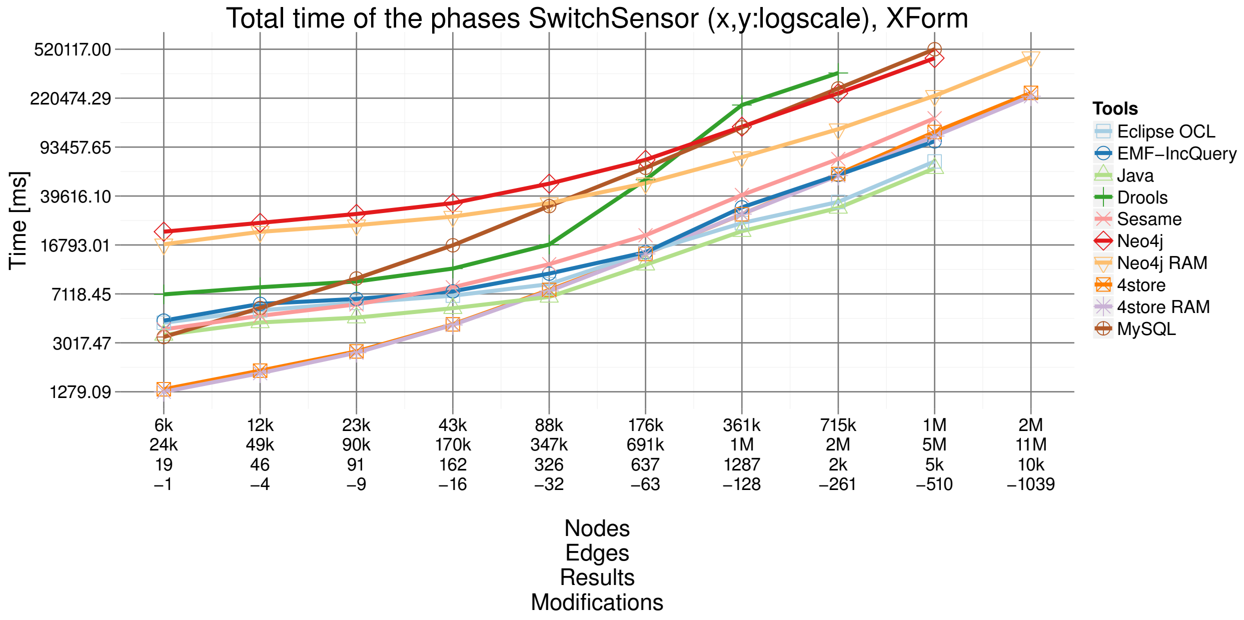


Figure 3.24: Total time for the SwitchSensor query in the Repair scenario.

3.3.5 Series of Edit times

Series of Edit Times (Inject Scenario)

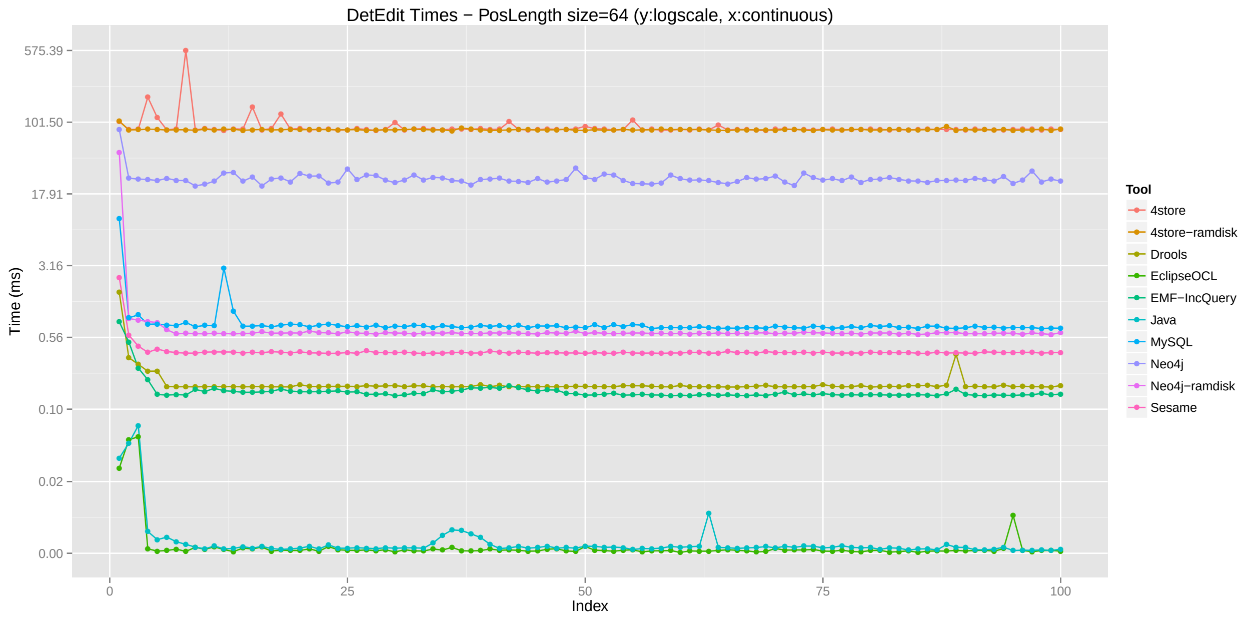


Figure 3.25: Series of edit times for the PosLength query in the Inject scenario.

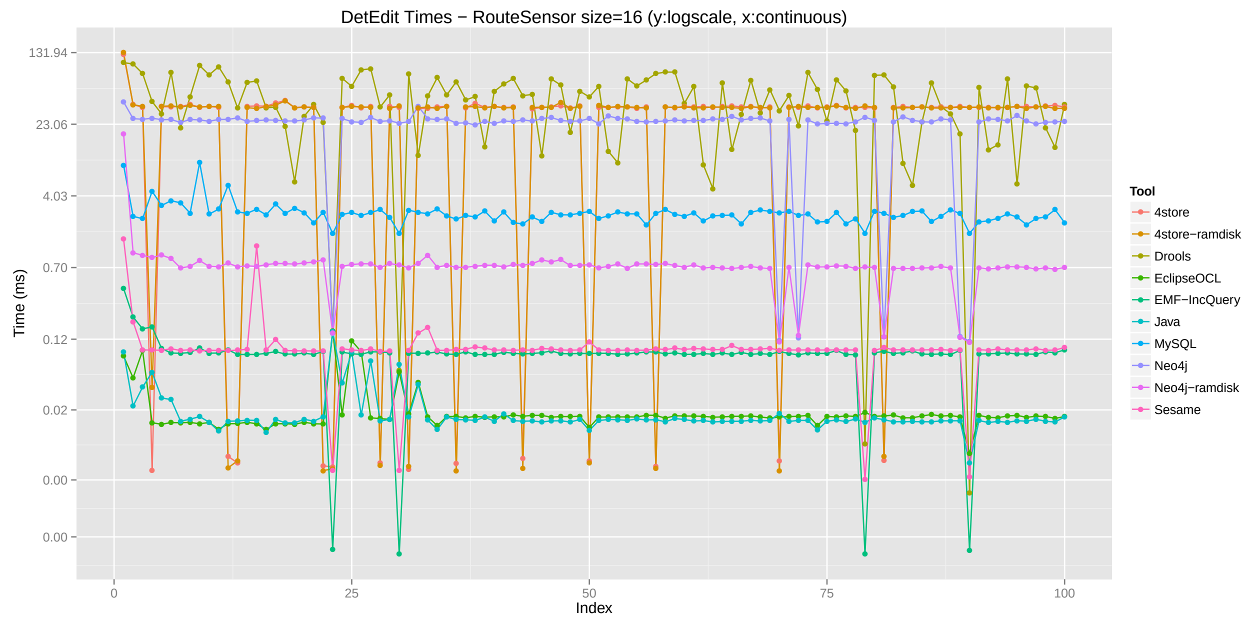


Figure 3.26: Series of edit times for the RouteSensor query in the Inject scenario.

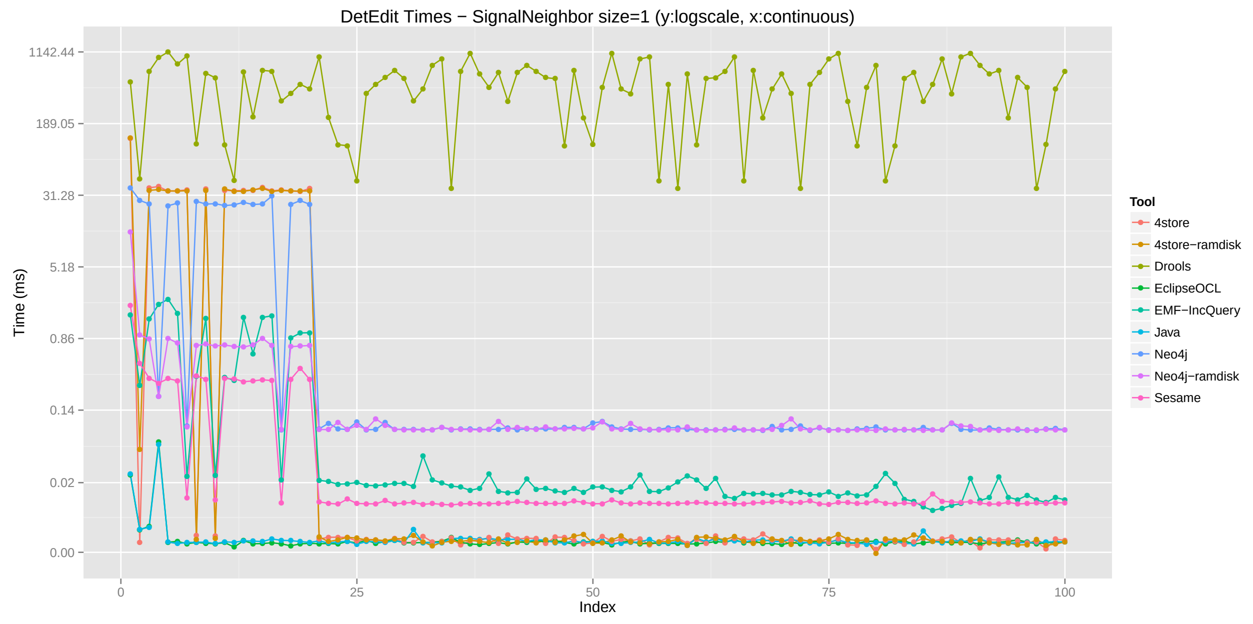


Figure 3.27: Series of edit times for the SemaphoreNeighbor query in the Inject scenario.

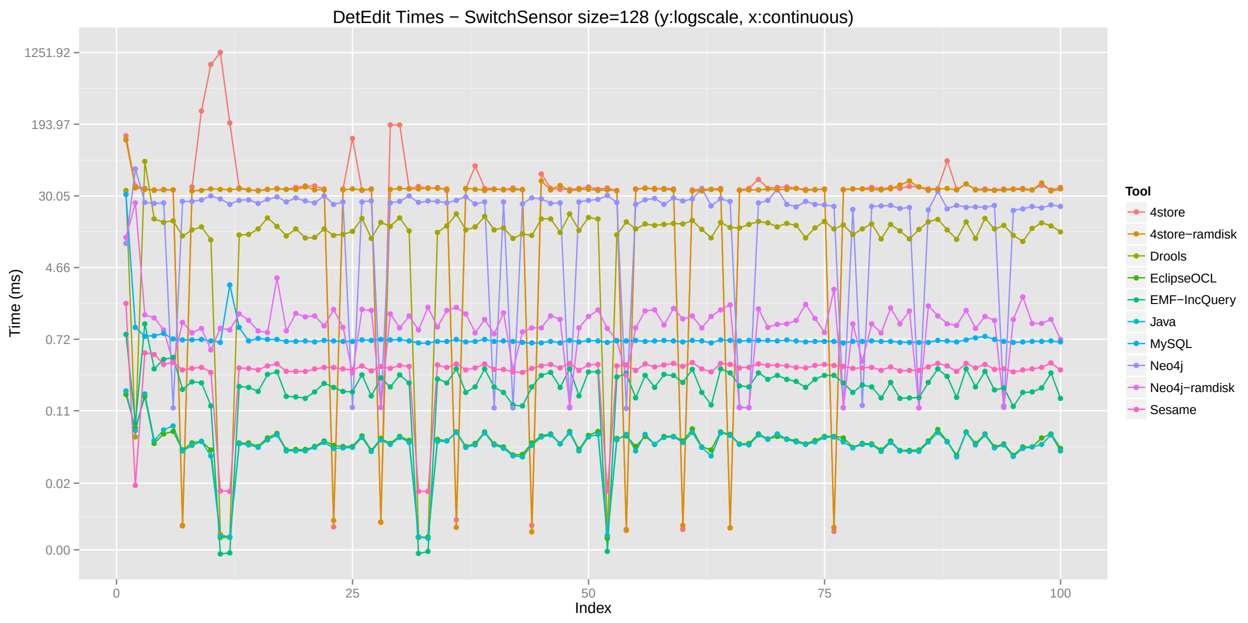


Figure 3.28: Series of edit times for the SwitchSensor query in the Inject scenario.

3.3.6 Series of Incremental Revaliation Times

Series of Incremental Revalidation Times (Inject Scenario)

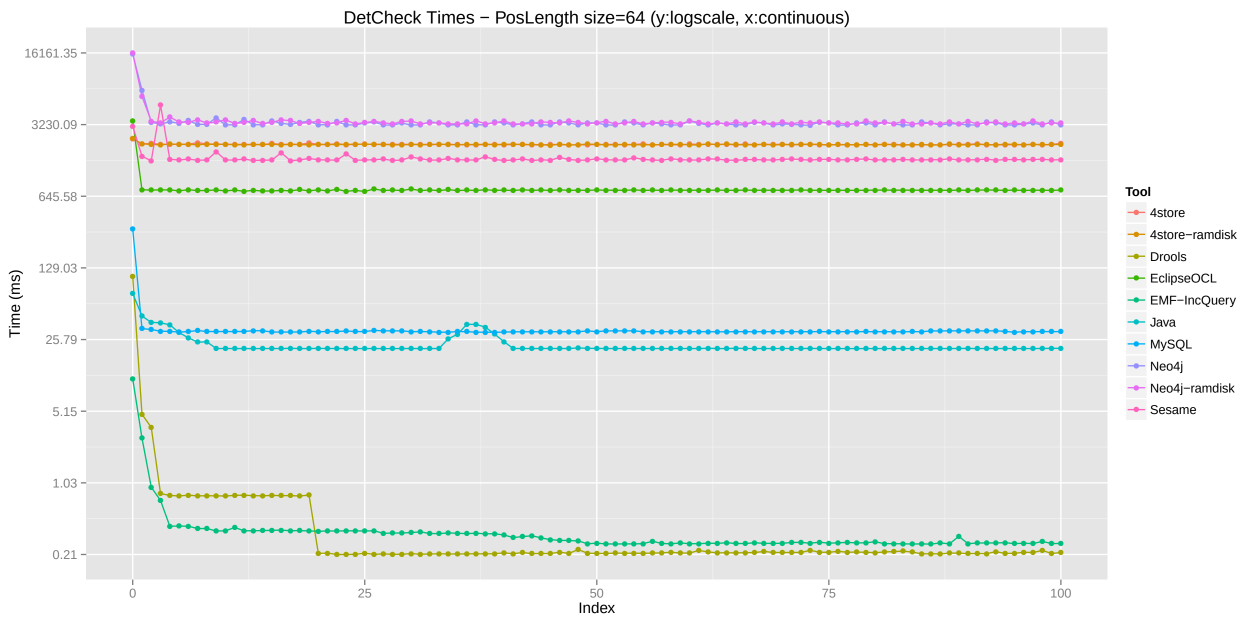


Figure 3.29: Series of incremental revalidation times for the PosLength query in the Inject scenario.

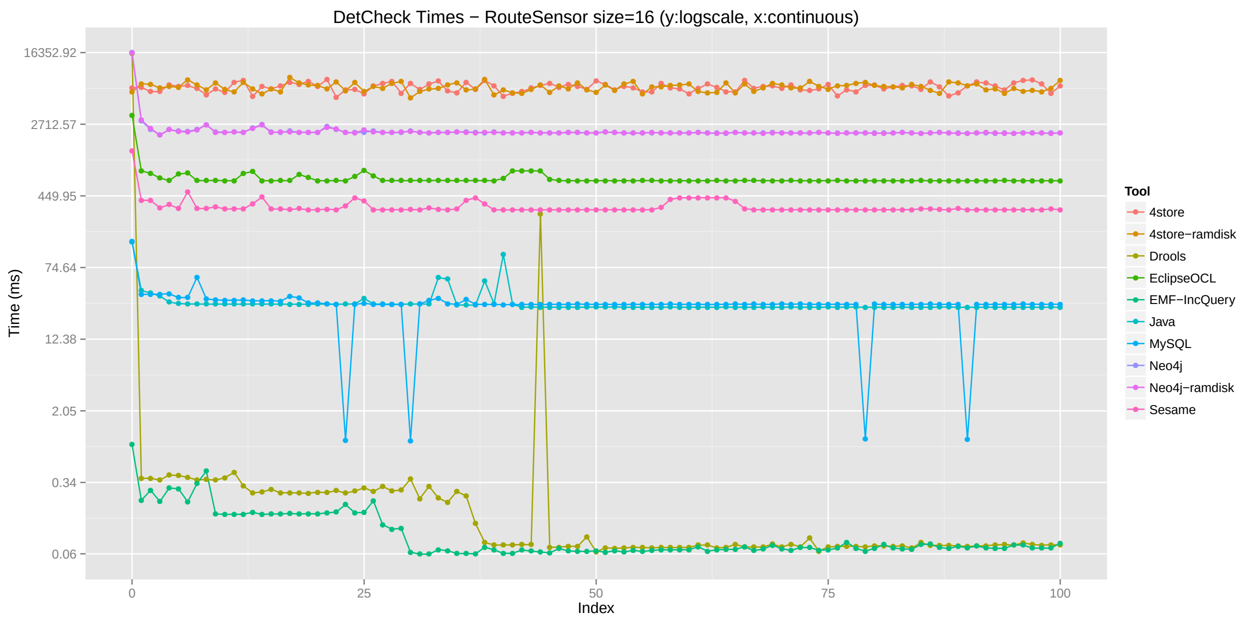


Figure 3.30: Series of incremental revalidation times for the RouteSensor query in the Inject scenario.

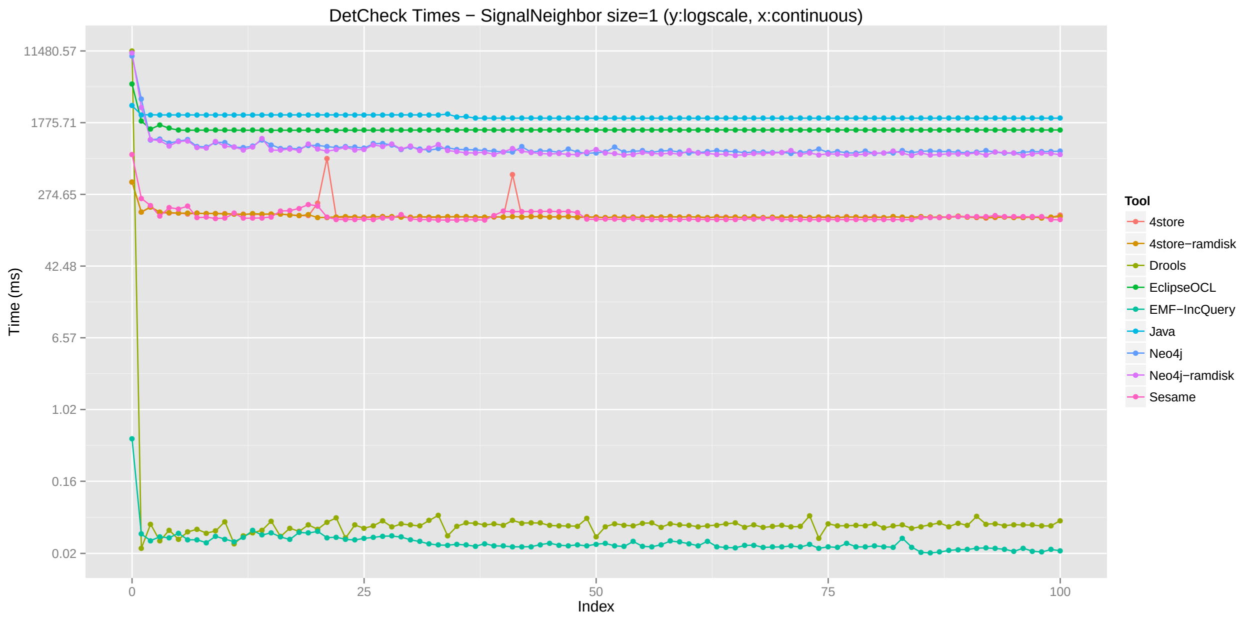


Figure 3.31: Series of incremental revalidation times for the SemaphoreNeighbor query in the Inject scenario.

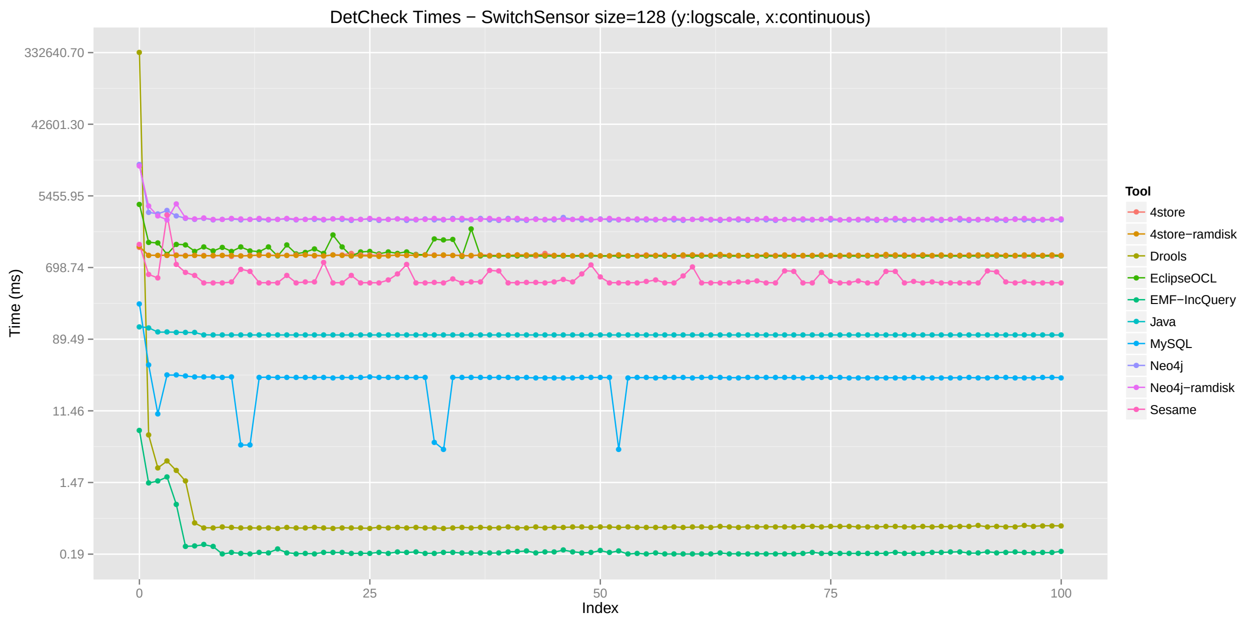


Figure 3.32: Series of incremental revalidation times for the SwitchSensor query in the Inject scenario.

3.4 Analysis of results

3.4.1 Batch validation

As it can be seen in the plots of Section 3.3.2, the overall execution time characteristics of all tools are similar. With the exception of the *SemaphoreNeighbor* query, even the (asymptotic) slopes are identical, which means that there is only a constant multiplier difference between the various tools. The *SemaphoreNeighbor* query proved to be significantly more complex than the others (as indicated by metrics evaluation results) as the slopes for some tools (Java, Eclipse OCL and Drools) are larger, indicating an exponential disadvantage compared to the fastest tools (e.g. Neo4j, EMF-INCQUERY and Sesame).

Taking a closer look at the dominantly constant-multiplier differences for queries *PosLength*, *SwitchSensor* and *RouteSensor*, the measured tools can be approximately grouped into three categories:

- The *fastest tools* are Eclipse OCL and EMF-INCQUERY, which is no surprise since they operate on in-memory structures and use non-trivial optimizations supported by their evaluation engines.
- Sesame, Java and Drools qualify as the *medium* of the spectrum, explained by the fact that they use in-memory structures but lack the optimization capabilities of EMF-INCQUERY and Eclipse OCL. This is somewhat surprising for Drools as it is advertised to incorporate optimization capabilities, yet this is not seen in our results.
- MySQL and Neo4j correspond to the *slower end*, possibly explained by their use of less optimized disk-oriented data structures and underoptimized query evaluation engines (especially

the Cypher engine in the case of Neo4j). The usage of RAM disks can alleviate the speed disadvantage only insignificantly.

IncQuery vs. OCL specific observations. From the *batch query* evaluation of the PosLength, SwitchSensor and RouteSensor queries (Section 3.3.2), it can be seen that EMF-INCQUERY performs similarly to Eclipse OCL. It is slightly faster for small models (2 s and 3 s respectively), but is slower for large models (up to 125 s and 78 s), where this 50% slowdown (once in the whole scenario) can be attributed to the initial (Rete) cache build.

However, for the more complex SemaphoreNeighbor query, it is observed that EMF-INCQUERY outperforms Eclipse OCL by a large margin solutions: it is noticeably faster for small modells (2 s and 4 s), and over 435k model elements OCL did not finish with the initial analysis in 12 minutes. This performance gain might be attributed to the more efficient (cached) enumeration of instances, and the possibility of backward navigation (with the help of auxiliary structures) on unidirectional references used by this query.

3.4.2 Incremental revalidation

As designed, this phase illustrates the characteristic difference between traditional, “batch” query engines and incremental tools clearly. *Total revalidation time* (shown on the X axis in the plots of Section 3.3.3) corresponds to the execution time of editing and re-checking operations, i.e. it approximates an incremental model transformation sequence for incremental tools and a batch transformation + batch query evaluation sequence for traditional tools.

The characteristic advantage of incremental tools (i.e. EMF-INCQUERY and Drools) is best seen on the Y axes of the plots, as these tools are typically 1-2 orders of magnitude (i.e. 10 to 100 times) faster than the rest. Since the plots include editing and result retrieval operations that scale linearly with model size, the overall low-order polynomial characteristics apply to all of the tools. The SemaphoreNeighbor query again highlights EMF-INCQUERY’s clear advantage when dealing with very complex queries: it beats all other tools by 3-5 orders of magnitude.

3.4.3 Highlights of Interest

Differences between the Inject and Repair scenarios. As noted in Section 2.1.5, the main difference between the Inject and Repair scenarios is the amount of model manipulation operations, which is significantly larger for the Repair scenario. The query result sets are also larger for the Repair scenario. By comparing corresponding plots, it is observed that the overall evaluation time is affected linearly by this difference (explained by the specification which requires a complete reading through the entire result list), meaning that all tools are capable of handling this efficiently.

Execution time differences can be huge. While the overall characteristics of all tools are similar (low order polynomial with a constant component), we have recorded a rather large variation of execution times (0.5–4 orders of magnitude difference). This confirms our expectation that model persistence, (de)serialization, query evaluation and model transformation technologies can have a huge overall impact on MDE performance.

RAM disks matter less than anticipated. Some measurements using 4store and Neo4j were performed with both SSD-based and RAM disk-based storage backends. Overall, as it is seen from total time measurements (Section 3.3.4), RAM disk-based variants were about twice as fast – which is less than we originally anticipated.

Hand-coded Java and MySQL are surprisingly fast in some cases. We created the hand-coded Java and MySQL implementations as baselines to which more sophisticated tools can be measured. While the advanced tools are the clear winners overall, the Java and MySQL baselines were surprisingly fast in several configurations. We interpret this as a sign that more sophisticated tools, and specifically MDE technologies still have a lot of performance enhancement potential to be unlocked.

JIT HotSpot optimizations kick in late. As seen in Section 3.3.5 and even better in Section 3.3.6, the execution time for certain operations executed in cycles can decrease dramatically over time. In the Java Runtime environment, this effect is best explained by Just-in-time (HotSpot) compiling by the Java Virtual Machine. Overall, it can be concluded that at least 30–50 complex operation iterations are advised to be executed in a benchmarking environment to obtain results that are relevant to long running software.

3.4.4 Threats to Validity

Internal threats

Mitigating measurement risks. We tried to mitigate *internal validity* threats by reducing the number of uncontrolled variables during measurements: a physical machine was used with all unnecessary software components turned off and every execution was isolated into a freshly initialized JVM.

Ensuring functional equivalence and correctness. The queries are semantically equivalent in the different query languages and the result sets are the same for every model. Our current measurements only loaded and executed a single query in each run. When loading multiple queries, query interference may change the results greatly. A more detailed evaluation of this issue is planned for the future.

Code reviews. Additionally, to ensure comparable results the created high-quality query implementations were reviewed: the OCL implementation by Ed Willink from the Eclipse OCL project, the usage of Impact Analyzer by Axel Uhl from the Impact Analyzer developer team. The graph patterns were written by the developers of EMF-INCQUERY. SPARQL code was reviewed by a semantic web expert. Drools rules were reviewed by the JBoss Drools developer team.

External threats

Metrics and generalizability of results. Considering *external validity*, the generalization of the results largely depends on how representative the metamodels, the models and the queries are compared to real use cases. In section 2.2 metrics were defined to describe complexity of models and queries, however comparing them to real world ones remains a future work.

Industrial relevance. The metamodel and the query specifications were motivated by an industrial case study, and the selected queries feature commonly used validation tasks such as attribute and reference checks or cycle detection. We tried to ensure that the instance models have a similar structure and distribution to other models by parameterizing the generation process based on our experience with other domains. To summarize, we believe that the train domain and the generated instance models represent other domain-specific languages and available instance models well.

3.5 Evaluation of Metrics

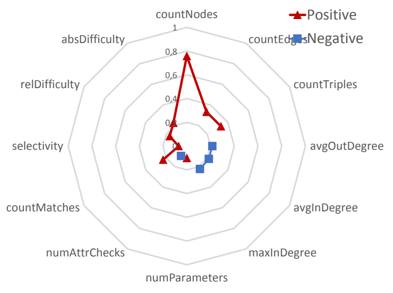
3.5.1 Method of Analysing Metrics

Our long-term goal is providing a catalog of reliable benchmark data that, based on metrics of the model and the queries, will allow the engineer to extrapolate the expected performance characteristics of various query technologies, which in turn will support making an informed decision on the solution to apply. In scope of this paper, we attempt to provide a necessary prerequisite with a narrower focus of investigation: finding out which model and query metrics are useful for predicting the performance of the various tools, over a wide range of different queries and model structures.

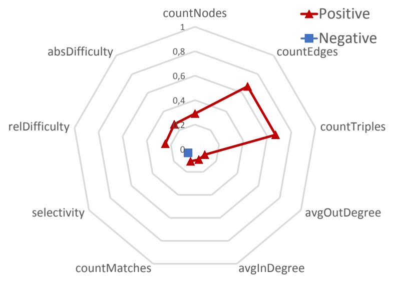
A given metric can only be a useful predictor of a certain performance indicator of a certain tool if they are strongly correlated. Therefore our analysis investigated correlations of metrics and performance indicators. Neither previous experience nor current measurement data supported a linear relationship between these variables; therefore we opted to abandon the commonly used Pearson correlation coefficient that is associated with linear regression. We rely instead on Kendall's τ rank correlation coefficient, ranging from -1 to $+1$; without assuming a linear model, τ characterises the degree to which it can be said that larger values of the metric correspond to larger values of the performance indicator.

Correlation coefficients may lead to incorrect conclusions if the sample of models and queries is too small. Therefore, whenever measuring an absolute value of τ, we additionally conducted the associated statistical test (τ-test) to decide whether the detected correlation between the variables is statistically significant ($p < 0.001$). Note that any such statistical result is conditional to the uniform sampling of the selected queries and models.

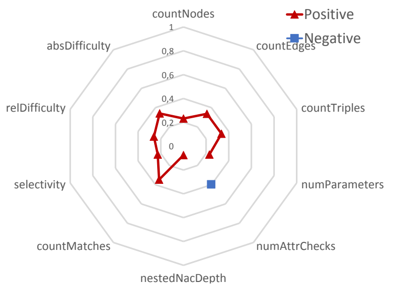
A limitation of the approach is that two strongly correlated variables (e.g. `countEdges` and `countTriples`) may show up as equally good predictors, but in reality they can not be used as two independent signals for predicting performance (as most triples in our models are edges). The simple correlation-based analysis presented here is intended as preliminary feature selection; we intend to follow up with redundancy-reducing feature selection techniques such as mRMR [23] that



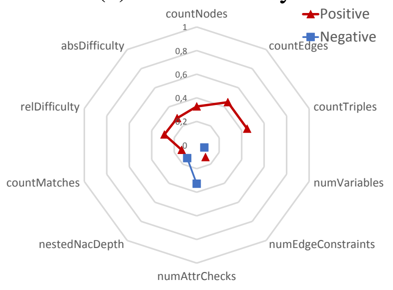
(a) Java Memory



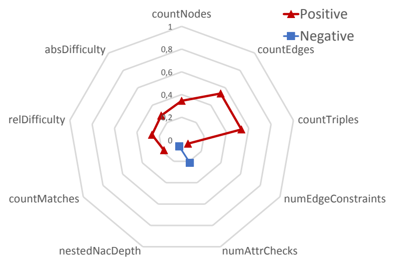
(b) Java Read time



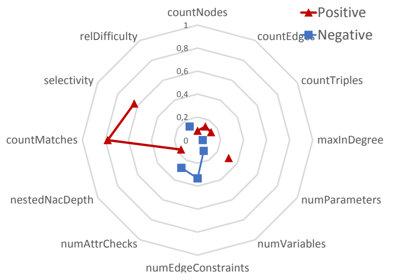
(c) Java Check time



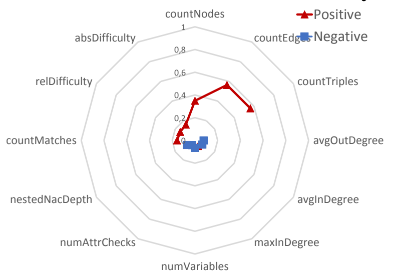
(d) EMF-IncQuery Memory



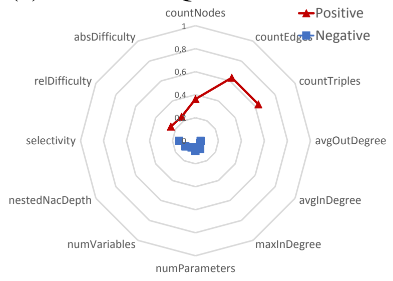
(e) EMF-IncQuery Read time



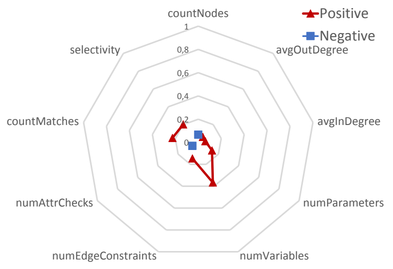
(f) EMF-IncQuery Check time



(g) Sesame Memory



(h) Sesame Read time



(i) Sesame Check time

Figure 3.33: $|\tau|$ of correlating ($p < 0.001$) metrics for each performance indicator.

can take into account which metrics convey independent information. Afterwards, predicting the best choice of technology based on the metric values is a problem of multivariate regression / classification, for which we plan to employ advanced machine learning techniques (e.g. ID3 [24] decision trees or MARS [17] regression model) in future work.

3.5.2 Metrics Colleration Results

For each tool performance indicator and each metric, we conducted Kendall's correlation test to see whether the data available is sufficient to form a statistically significant support of correlation between the performance indicator and the metric. For metrics that were found to correlate, the absolute value of Kendall's τ correlation coefficient is displayed on a spider chart specific to the performance indicator of the tool (see Figure 3.33); positive correlation values are displayed as red triangles, while negative ones as blue squares.

Evaluation

Consistently with previously published results, the data shows that model size is a strong predictor of both model loading time and memory consumption, regardless of the technology.

Tool-specific Observations However, check times show a more diverse picture. The check times for the Java implementation (being a dominantly search-intensive approach) are additionally correlated with the query-on-model metrics as well, with the strongest correlation shown by the *absDifficulty* metric.

Interestingly, the best Sesame check time predictor turned out to be the number of pattern variables, and there is no significant correlation with any direct model metrics.

Check times in EMF-INCQUERY are very strongly correlated to the number of matches – which is to be expected of an incremental tool whose check phase consists of just enumerating the cached match set. As the incremental indexes are constructed during the load time, the model-on-query metrics become correlated with EMF-INCQUERY read time. It can also be highlighted that EMF-INCQUERY seems not to be sensitive towards the "difficulty" of the query (in any phase) or the model size (during the check phase) due to the very small correlations with corresponding metrics.

Metrics Overall, it can be said that model-only metrics are useful in predicting the performance of model persistence operations. However, query-based and combined metrics (such as our proposed *abs-* and *relDifficulty*) are necessary to provide a more thorough picture. Note that since only statistically significant correlations are included, a low magnitude correlation does not necessarily mean a measurement error. It is possible that there is a true link between the two variables, but the τ value is lowered by other metrics that strongly influence the performance indicator.

3.5.3 Threats to Validity

Regarding the technological foundations and methodology of our measurements, the most important threats to validity stem from *time measurement uncertainty* and distortions due to transient effects such as *garbage collection* in the JVM and *thrashing* due to heap size exhaustion. Such effects were mitigated by using the most accurate Java time measurement method (`System.nanoTime`), allocating as much heap as possible, and using a timeout mechanism to identify and exclude cases affected by thrashing from the results. Additionally, it is also possible that there is sampling bias in our choice of models and metrics; we believe that this is sufficiently mitigated by our systematic choice of model generation strategies and the design principles of the queries. To improve the magnitude of correlation we increased the sample size by running the benchmarks ten times.

Chapter 4

Summary

In this tech report, we presented the *Train Benchmark*, an extensible framework for the definition and execution of benchmark scenarios for modeling tools. The framework supports the construction of benchmark test sets that specify the metamodel, instance model generation, queries and transformations, result collection and processing, and metric evaluation logic that are intended to provide an end-to-end solution. Additionally, we also presented a concrete test case set corresponding to the the original Train Benchmark implementation of [30] and its metric-focused extensions in [21].

In addition to the results presented in previous academic papers, the tech report contains a comprehensive set of measurement results comparing 7 different tools from three technological domains (Ecore, RDF/SPARQL, property graphs). These results allow for intra-domain and inter-domain tool comparison and detailed execution time characteristics analysis.

As future work, we plan to incorporate further tools (such as Egyed's OCL Impact Analyzer, Epsilon, ATL, other NoSQL databases using the Tinkerpop API and the Gremlin language). Additionally, work has begun to create a new benchmark case based on source code modernization base study in MONDO.

Appendix A

Appendix

A.1 Rete Networks for the Queries of the Train Benchmark

The Rete algorithm uses *tuples* to represent the vertices (along with their properties), edges and subgraphs in the graph. The algorithm defines an asynchronous network of communicating nodes. The Rete networks has three layers:

- The top level consists of *input nodes* which are type-instance indexers for the model. Each input node indexes a certain vertex type or edge label. The actual graph element is indicated by a vertex/edge icon in the upper right corner of the Rete node.
- *Worker nodes* perform a transformation on the output of their parent node(s) and propagate the results. Partial query results are represented in tuples and stored in the memory of the worker node thus allowing for incremental query reevaluation.
- The bottom level consist of *production nodes* which are terminators that provide an interface for fetching the results of the query and the changes introduced by the latest transformation.

In Figure A.1–Figure A.4, we present possible Rete networks layouts for the queries in the Train Benchmark.

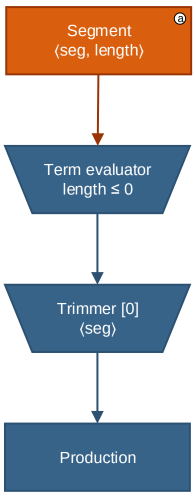


Figure A.1: The Rete network for the PosLength query.

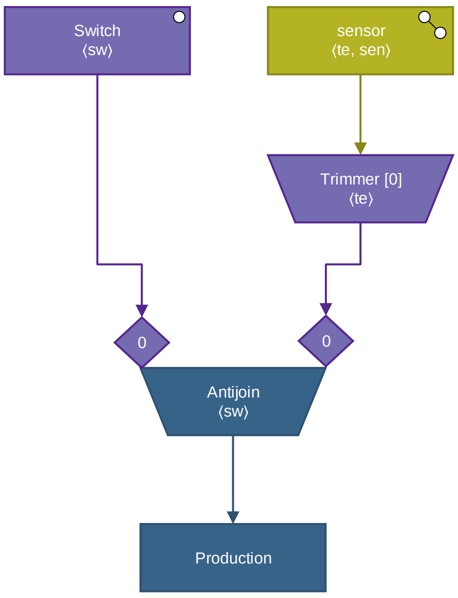


Figure A.2: The Rete network for the SwitchSensor query.

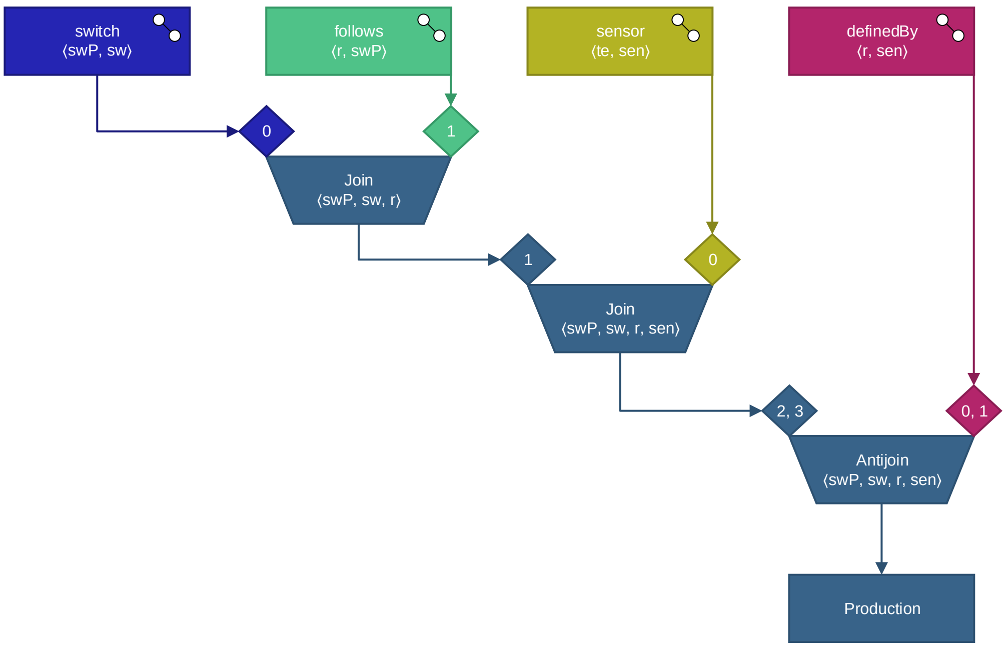


Figure A.3: The Rete network for the RouteSensor query.

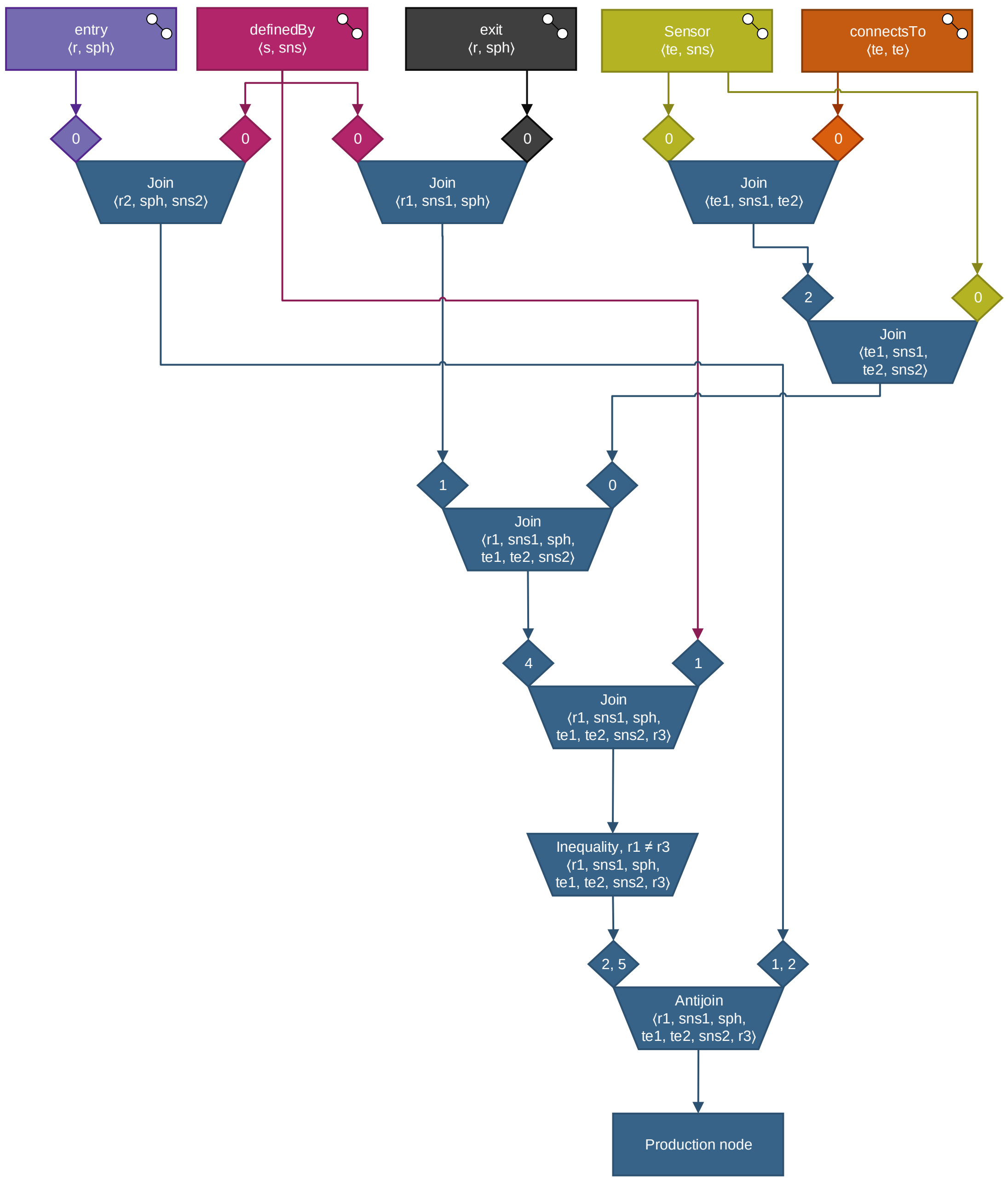


Figure A.4: The Rete network for the SemaphoreNeighbor query.

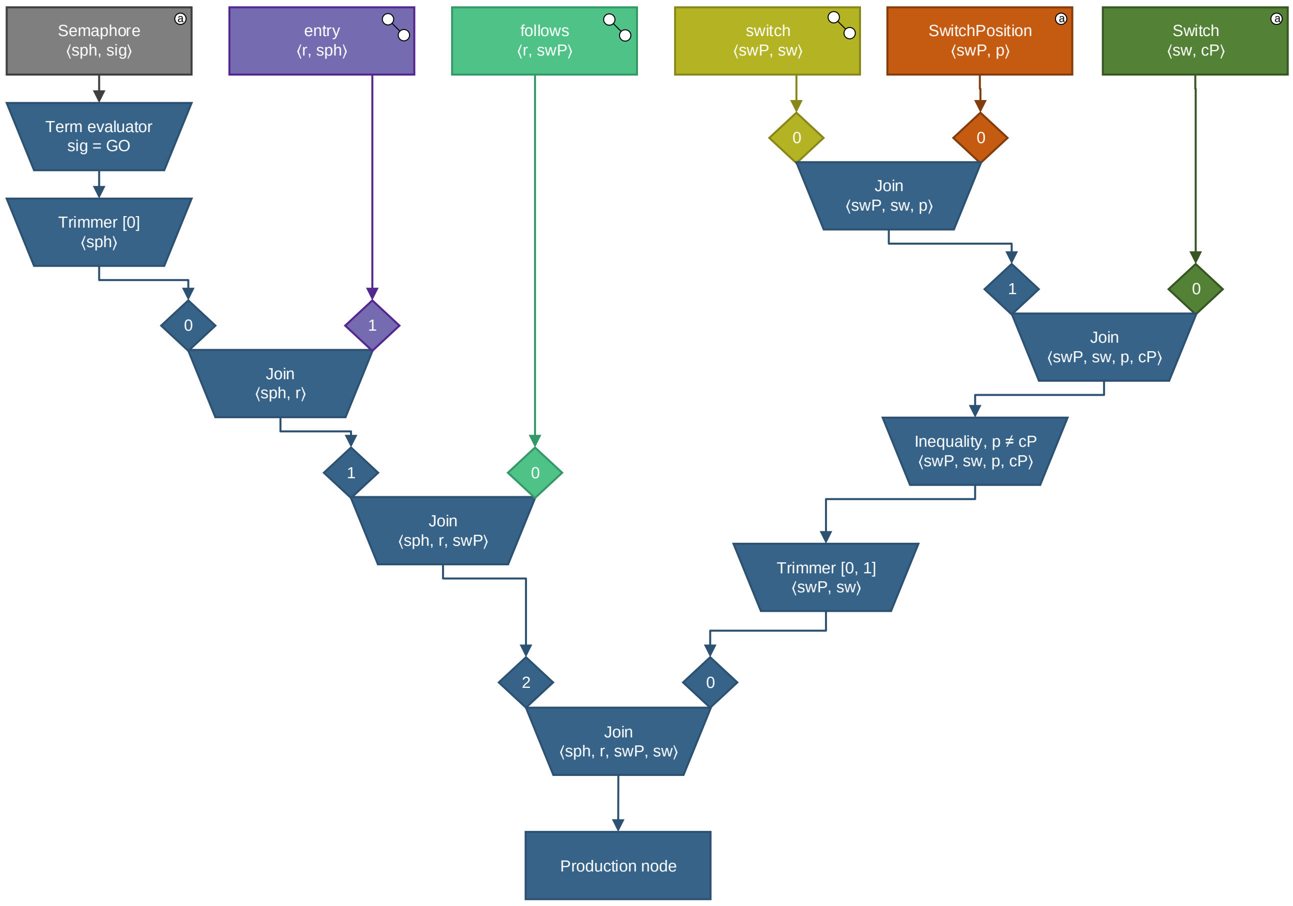


Figure A.5: The Rete network for the SwitchSet query.

Bibliography

[1] Apache Maven. `http://maven.apache.org`.

[2] MySQL. `http://www.mysql.com/`.

[3] Neo Technology: Neo4j. `http://neo4j.org/`.

[4] Redhat Drools Expert. `http://www.jboss.org/drools`.

[5] Model-based generation of tests for dependable embedded systems, 7th eu framework programme. `http://http://mogentes.eu/`, 2011.

[6] The Train Benchmark Website. `https://incquery.net/publications/trainbenchmark/full-results`, 2013.

[7] The R project for statistical computing. `http://www.r-project.org/`, 2014.

[8] Thara Angskun, George Bosilca, and Jack Dongarra. Self-healing in binomial graph networks. In *Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems - Volume Part II*, OTM'07, pages 1032–1041, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] Albert-László Barabási and Zoltán N. Oltvai. Network biology: understanding the cell's functional organization. *Nat Rev Genet*, 5(2):101–113, February 2004.

[10] Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Proc. 4th International Conference on Graph Transformations, ICGT 2008*, volume 5214 of *Lecture Notes in Computer Science*, pages 396–410. Springer, Springer, 2008. Acceptance rate: 40%.

[11] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. Incremental evaluation of model queries over EMF models. In *Model Driven Engineering Languages and Systems, 13th International Conference, MODELS'10*. Springer, Springer, 10/2010 2010. Acceptance rate: 21%.

[12] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A Graph Query Language for EMF models. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations, ICMT 2011*, volume 6707 of *LNCS*, pages 167–182. Springer, Springer, 2011.

[13] Christian Bizer and Andreas Schultz. The Berlin SPARQL benchmark. *International Journal on Semantic Web & Information Systems*, 5(2):1–24, 2009.

[14] Eclipse Model Development Tools Project. Eclispe OCL website, 2011. `http://www.eclipse.org/modeling/mdt/?project=ocl`.

[15] P. Erdős and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5:17–61, 1960.

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

[17] Jerome H. Friedman. Multivariate adaptive regression splines. *Ann. Statist.*, 19(1):1–141, 1991. With discussion and a rejoinder by the author.

[18] Olaf Görlitz, Matthias Thimm, and Steffen Staab. SPLODGE: Systematic generation of SPARQL benchmark queries for Linked Open Data. In Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, JosianeXavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web – ISWC 2012*, volume 7649 of *Lecture Notes in Computer Science*, pages 116–132. Springer Berlin Heidelberg, 2012.

[19] 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.

[20] 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.

[21] Benedek Izsó, Zoltán Szatmári, Gábor Bergmann, Ákos Horváth, and István Ráth. Towards Precise Metrics for Predicting Graph Query Performance. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 412–431, Silicon Valley, CA, USA, 11/2013 2013. IEEE, IEEE. Acceptance Rate: 23%.

[22] Object Management Group. *Object Constraint Language Specification (Version 2.3.1)*, 2012. `http://www.omg.org/spec/OCL/2.3.1/`.

[23] H. Peng, Fulmi Long, and C. Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(8):1226–1238, 2005.

[24] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, March 1986.

[25] Arend Rensink. The groove simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance*, pages 479–485. Springer, 2004.

[26] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP2Bench: A SPARQL performance benchmark. In *Proc. of the 25th International Conference on Data Engineering*, pages 222–233, Shanghai, China, 2009. IEEE.

[27] Jakub Stárka, Martin Svoboda, and Irena Mlynkova. Analyses of RDF triples in sample datasets. In *COLD*, 2012.

[28] The Eclipse Project. Eclipse Modeling Framework. `http://www.eclipse.org/emf/`.

[29] Transaction Processing Performance Council (TPC). TPC-C Benchmark, 2010.

[30] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. EMF-IncQuery: An Integrated Development Environment for Live Model Queries. *Science of Computer Programming*, 2014. Accepted.

[31] Batagelj Vladimir and Brandes Ulrik. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 04 2005.

[32] World Wide Web Consortium. Resource Description Framework (RDF). `http://www.w3.org/standards/techs/rdf/`.

[33] World Wide Web Consortium. SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`.