Train Benchmark Case: an EMF-INCQUERY Solution*

Gábor Szárnyas Márton Búr István Ráth
Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
szarnyas@mit.bme.hu, marton.bur@inf.mit.bme.hu, rath@mit.bme.hu

This paper presents a solution for the Train Benchmark Case of the 2015 Transformation Tool Contest, using EMF-INCQUERY.

1 Introduction

This paper describes a solution for the TTC 2015 Train Benchmark Case [6]. The source code of the solution is available as an open-source project.[1] There is also a SHARE image available.[2]

2 EMF-INCQUERY

Automated model transformations are frequently integrated with modeling environments, requiring both high performance and a concise programming interface to support software engineers. The objective of the EMF-INCQUERY [2] framework is to provide a declarative way to define queries over EMF models. EMF-INCQUERY extended the pattern language of VIATRA2 with new features (including transitive closure, role navigation, match count) and tailored it to EMF models [4]. EMF-INCQUERY is developed with a focus on *incremental query evaluation*, however, the most recent version is also capable of evaluating queries with a *local search-based* algorithm.

2.1 Incremental Pattern Matching

EMF-INCQUERY uses the Rete algorithm [1] to perform incremental pattern matching. The Rete algorithm uses tuples to represent the model objects, attributes, references and partial matches in the model. The algorithm defines an asynchronous network of communicating nodes. The network consists of three types of nodes. Input nodes are responsible for indexing the model by type, i.e. they store the appropriate tuples for the objects and references. They are also responsible for producing the update messages and propagating them to the worker nodes. 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. Production nodes are terminators that provide an interface for fetching the results of the query and the changes introduced by the latest transformation. Moreover, parallelization possibilities of the algorithm were already investigated in [3].

*This work was partially supported by the MONDO (EU ICT-611125) project and Red Hat Inc.

[1] https://github.com/FTSRG/trainbenchmark-ttc

[2] http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=ArchLinux64_TrainBenchmark-EIQ.vdi

© G. Szárnyas et al.
This work is licensed under the
Creative Commons Attribution License.

The incremental pattern matcher provides quick reevaluation for complex queries. However, it does so at the expense of high memory consumption as the partial results are stored in the Rete network.

2.2 Local Search-Based Pattern Matching

Local search-based pattern matching (LS) is commonly used in graph transformation tools. Along with the incremental query engine, EMF-INCQUERY also provides a local search-based pattern matcher.

The matching process consists of four steps. (1) At first, in a preprocessing step the patterns are *normalized*: the constraint set is minimized, variables that are always equal are unified and positive pattern calls are flattened. These normalized patterns are evaluated by (2) the *query planner*, using a specified cost estimation function to provide search plans: totally ordered lists of search operations used to ensure that the constraints from the pattern definition hold. From a single pattern specification multiple search plans can be derived, thus pattern matching includes (3) *plan selection* based on the input parameter binding and model-specific metrics. Finally, (4) *the search plan is executed* by a plan interpreter evaluating the different operations of the plans. If an operation fails, the interpreter backtracks; if all operations are executed successfully, a match is found.

Compared to the incremental query engine, the search-based algorithm requires less memory [7] and is therefore capable of performing queries on larger models if there is not enough memory available for the incremental engine.

2.3 Defining the Pattern Matching Strategy

Currently, the pattern matching strategy has to be determined by the developer by specifying the query backend of EMF-INCQUERY. Developing a hybrid pattern matching engine is subject to future work. This will allow the user to use annotations to define the evaluation strategy for each pattern. There are also plans to develop an adaptive query engine (Section 5).

2.4 Pattern Match Representation

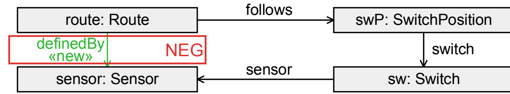
For each query, EMF-INCQUERY generates a set of utility classes. These classes store the model objects in the match and provide a convenient interface for reading and transforming the matches. These classes are used for implementing the transformation operations (Section A.1).

3 Solution

The case defines a well-formedness validation scenario set in the domain of railway systems [6]. The case provides a synthetic instance model generator which is capable of generating models of various sizes. For the solution, we used the metamodel defined in the case description without any modifications or extensions.

The solution was developed in the Eclipse IDE. For setting up the development environment, please refer to the readme file. The projects are not tied to the Eclipse environment and can be compiled with the Apache Maven build automation tool. This offers a number of benefits, including portability and the possibility of continuous integration. The solution is written in Java 7. The patterns are defined in INCQUERY Pattern Language (IQPL) [4].

3.1 Example Query: RouteSensor



We describe the implementation of the RouteSensor query in detail. The other queries and transformations are implemented in a similar manner. The implemented application uses the Java classes generated by EMF-INCQUERY and the hand-coded transformation logic introduced below. First it finds the matches of the queries, then the corresponding transformation step is applied for each match. The code of patterns and the transformation definitions are listed in Section A.1.

```
1 pattern routeSensor(route, sensor, switchPosition, sw)
2 {
3   Route.follows(route, switchPosition);
4   SwitchPosition.^switch(switchPosition, sw);
5   TrackElement.sensor(sw, sensor);
6   neg find definedBy(route, sensor);
7 }
8
9 pattern definedBy(route, sensor)
10 {
11   Route.definedBy(route, sensor);
12 }
```

Listing 1: Pattern of the RouteSensor query.

The RouteSensor query looks for sensors that are connected to a switch, but the sensor and the switch are not connected to the same route. The query in IQPL is listed in Listing 1. The positive conditions are defined by using the appropriate classes and references, while the negative application condition (NAC) is defined as a negative find operation (`neg find`) for a separate query.

During the repair operation, for the selected matches, the missing definedBy edge is inserted by connecting the route to the sensor. The Java transformation code implementing the transformation is listed in Listing 2. The transformation uses the match object returned by EMF-INCQUERY.

```
1 public void transform(final Collection<Object> matches) {
2   for (final Object match : matches) {
3     final RouteSensorMatch rsm = (RouteSensorMatch) match;
4     rsm.getRoute().getDefinedBy().add(rsm.getSensor());
5   }
6 }
```

Listing 2: Transformation of the RouteSensor query.

3.2 Query Evaluation Strategies for the RouteSensor Pattern

We use the RouteSensor query to provide an overview of the various query evaluation strategies used in EMF-INCQUERY.

3.2.1 Incremental Evaluation

The Rete network derived from the RouteSensor query is shown in Figure 7. For the sake of clarity, we simplified the Rete network by removing some implementation-specific details. The evaluation in the Rete network starts with the *input nodes* (switch, follows, sensor, definedBy), which are indexing the model by collecting the appropriate tuples. The *worker nodes* are responsible for performing the relational operations, *join* and *antijoin* in this case. The join nodes have a pair of tuple masks (e.g. $\langle 2,3 \rangle$ and $\langle 0,1 \rangle$) to determine the attributes used in the join operation. The match set of the pattern is stored in the *production node*.

3.2.2 Local Search-Based Evaluation

The search plan generated for evaluating the RouteSensor query is shown in Figure 8. This screenshot is taken from the *Local Search Debugger* view of EMF-INCQUERY. The search plan is presented in the upper-left part, while the found matches with the variable substitutions are shown below the search plan in a table viewer. The Zest-based graph viewer in the right visualises a match based on the selection of the table viewer.

4 Evaluation

In this section, we present the benchmark environment and evaluate the results.

4.1 Benchmark Environment

The benchmarks were performed on a 64-bit Ubuntu Server 14.04 virtual machine deployed on a private cloud. The machine used a quad-core 2.50 GHz Xeon L5420 processor and 16 GB of memory. We used Oracle JDK 8 and set the available heap memory to 15 GB.

4.2 Benchmark Results

To present the results, we use the reporting framework of the Train Benchmark. The framework generates plots to visualise the execution time of the phases defined in the benchmark. The plots showing each query are included in Section A.2. On each plot, the x axis shows the problem size, i.e. the size of the instance model, while the y axis shows the aggregated execution time of a certain phases, measured in milliseconds. Both axes use logarithmic scale.

4.2.1 Benchmark Results for the RouteSensor Query

For the sake of conciseness, we only discuss the results for the RouteSensor query in detail.

The results for the *batch validation* are shown in Figure 1. The results suggest that—given enough memory—both the incremental and the local search-based (LS) strategies are able to run the query and the transformation for the largest model. The *first validation* takes consistently longer for the incremental strategy as for the LS strategy. This is caused by the fact that the incremental strategy builds the Rete algorithm during the read phase. However, the difference is small as the *first validation* time largely consists of deserializing the EMF model.

The execution times of the *revalidation* are shown in Figure 2. The execution time of the incremental strategy linearly correlates with the size of the change set. This implies that for a *fixed* change set, the incremental strategy is able to perform the transformation in constant time, while execution time for the LS strategy correlates with the model size. For the *proportional* change set, the revalidation time is a low-degree polynomial of the model size for both strategies, however, it is an order of magnitude faster for the incremental strategy than for the LS.

4.3 Comparison of the Query Evaluation Strategies

Section A.2 shows the detailed results for all queries and both evaluation strategies. In the *first validation* (Figure 3 and Figure 5), the evaluation strategies show similar performance characteristics as both have

to compute the complete result set of the query. The execution times for both strategies show that the most complex query is SemaphoreNeighbor, while the simplest one is SwitchSensor.

As expected, the execution times of the *revalidation* are different for the two strategies. Figure 4 shows that for the incremental strategy the execution time correlates with the size of the match set (instead of the model size). This can be observed when comparing the execution times for the *fixed* and the *proportional* change sets. Figure 6 shows that the execution time for the LS strategy is determined by the model size and is not affected by the size of the change set.

These results imply that the optimal evaluation strategy depends on the specific workload profile. If there is enough memory available and the transformations operate on a small amount of model elements, it is recommended to use the incremental strategy. If the transformations often change a large proportion of the model elements, the LS strategy is recommended.

5 Summary and Future Work

The paper presented a solution for the Train Benchmark case of the 2015 Transformation Tool Contest.

There is ongoing work to develop a hybrid query engine [5] for EMF-INCQUERY. This will allow the user to use annotations on the patterns for specifying the desired query evaluation strategy. There are also plans to develop an adaptive query engine which will use query optimisation heuristics to determine the appropriate strategy based on the query, the model and the available resources.

Acknowledgements. The authors would like to thank Zoltán Ujhelyi for providing valuable insights into EMF-INCQUERY.

References

[1] Gábor Bergmann (2013): *Incremental Model Queries in Model-Driven Design*. Ph.D. dissertation, Budapest University of Technology and Economics, Budapest. Available at `http://home.mit.bme.hu/~bergmann/download/phd-thesis-bergmann.pdf`.

[2] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh & András Ökrös (2010): *Incremental Evaluation of Model Queries over EMF Models*. In: *MODELS*, Springer, Springer, doi:http://dx.doi.org/10.1007/978-3-642-16145-2_6.

[3] Gábor Bergmann, István Ráth & Dániel Varró (2009): *Parallelization of Graph Transformation Based on Incremental Pattern Matching*. *Electronic Communications of the EASST, Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques* 18. Available at `http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/265/249`.

[4] Gábor Bergmann, Zoltán Ujhelyi, István Ráth & Dániel Varró (2011): *A Graph Query Language for EMF Models*. In: *Theory and Practice of Model Transformations, Fourth Intl. Conf.*, *LNCS* 6707, Springer.

[5] Ákos Horváth, Gábor Bergmann, István Ráth & Dániel Varró (2010): *Experimental Assessment of Combining Pattern Matching Strategies with VIATRA2*. *International Journal on Software Tools for Technology Transfer* 12(3-4), pp. 211–230, doi:10.1007/s10009-010-0149-7. Available at `http://dx.doi.org/10.1007/s10009-010-0149-7`.

[6] Gábor Szárnyas, Oszkár Semeráth, István Ráth & Dániel Varró (2015): *The TTC 2015 Train Benchmark Case for Incremental Model Validation*. In: *8th Transformation Tool Contest (TTC 2015)*.

[7] Zoltán Ujhelyi, Gábor Szőke, Ákos Horváth, Norbert István Csiszár, László Vidács, Dániel Varró & Rudolf Ferenc (2015): *Performance Comparison of Query-based Techniques for Anti-Pattern Detection*. *Information and Software Technology*, doi:10.1016/j.infsof.2015.01.003. In press.

A Appendix

A.1 Patterns and Transformations

A.1.1 PosLength

```
1 pattern posLength(segment)
2 {
3   Segment.length(segment, length);
4   check(length <= 0);
5 }
```

Listing 3: Pattern of the PosLength query.

```
1 public void transform(final Collection<Object> matches) {
2   for (final Object match : matches) {
3     final RouteSensorMatch rsm = (RouteSensorMatch) match;
4     rsm.getRoute().getDefinedBy().add(rsm.getSensor());
5   }
6 }
```

Listing 4: Transformation of the PosLength query.

A.1.2 SwitchSensor

```
1 pattern switchSensor(sw)
2 {
3   Switch(sw);
4   neg find hasSensor(sw);
5 }
6
7 pattern hasSensor(sw)
8 {
9   TrackElement.sensor(sw, _);
10 }
```

Listing 5: Pattern of the SwitchSensor query.

```
1 public void transform(final Collection<Object> matches) {
2   for (final Object match : matches) {
3     final SwitchSensorMatch ssm = (SwitchSensorMatch) match;
4     final Sensor sensor = RailwayFactory.eINSTANCE.createSensor();
5     ssm.getSw().setSensor(sensor);
6   }
7 }
```

Listing 6: Transformation of the SwitchSensor query.

A.1.3 SwitchSet

```
1 pattern switchSet(semaphore, route, switchPosition, sw)
2 {
3   Route.entry(route, semaphore);
4   Route.follows(route, switchPosition);
5   SwitchPosition.^switch(switchPosition, sw);
6
7   Semaphore.signal(semaphore, ::GO);
8   SwitchPosition.position(switchPosition, swPP);
9   Switch.currentPosition(sw, swCP);
10
```

```
11   swPP != swCP;
12 }
```

Listing 7: Pattern of the SwitchSet query.

```
1 public void transform(final Collection<Object> matches) {
2   for (final Object match : matches) {
3     final SwitchSetMatch ssm = (SwitchSetMatch) match;
4     ssm.getSw().setCurrentPosition(ssm.getSwitchPosition().getPosition());
5   }
6 }
```

Listing 8: Transformation of the SwitchSet query.

A.1.4 RouteSensor

The RouteSensor query is discussed in detail in Section 3.1.

A.1.5 SemaphoreNeighbor

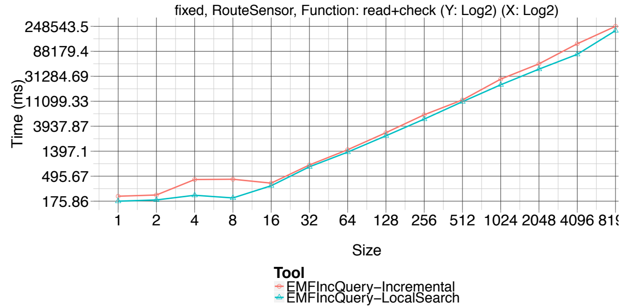
```
1 pattern semaphoreNeighbor(semaphore, route1, route2, sensor1, sensor2, te1, te2)
2 {
3   Route.exit(route1, semaphore);
4   Route.definedBy(route1, sensor1);
5   TrackElement.sensor(te1, sensor1);
6   TrackElement.connectsTo(te1, te2);
7   TrackElement.sensor(te2, sensor2);
8   Route.definedBy(route2, sensor2);
9   neg find entrySemaphore(route2, semaphore);
10
11   route1 != route2;
12 }
13
14 pattern entrySemaphore(route, semaphore)
15 {
16   Route.entry(route, semaphore);
17 }
```

Listing 9: Pattern of the SemaphoreNeighbor query.

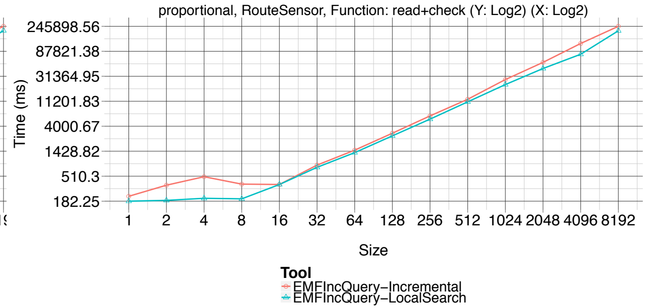
```
1 public void transform(final Collection<Object> matches) {
2   for (final Object match : matches) {
3     final SemaphoreNeighborMatch snm = (SemaphoreNeighborMatch) match;
4     snm.getRoute2().setEntry(snm.getSemaphore());
5   }
6 }
```

Listing 10: Transformation of the SemaphoreNeighbor query.

A.2 Detailed Benchmark Results

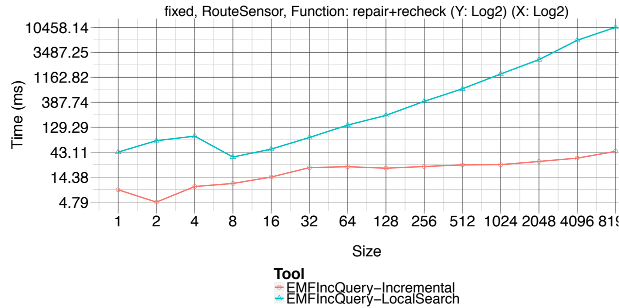


(a) Fixed change set

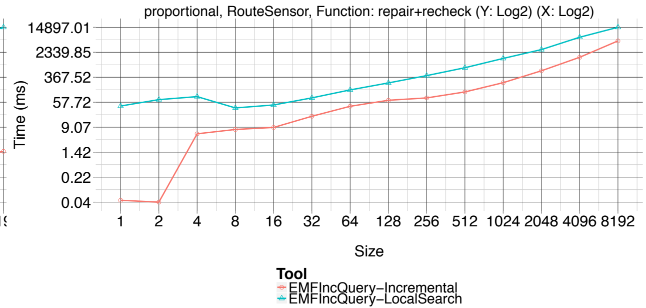


(b) Proportional change set

Figure 1: First validation times for the RouteSensor query.

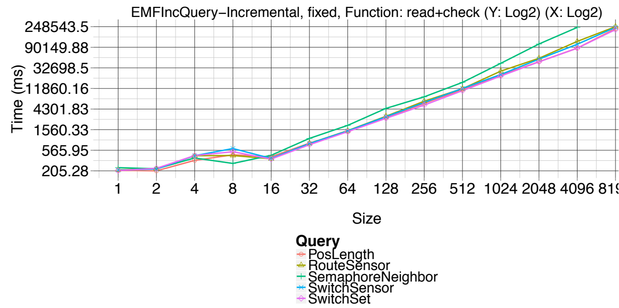


(a) Fixed change set

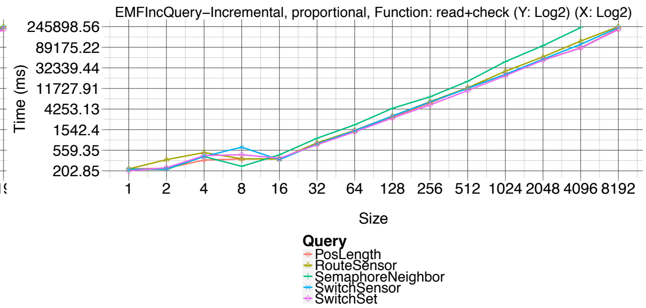


(b) Proportional change set

Figure 2: Revalidation times for the RouteSensor query.

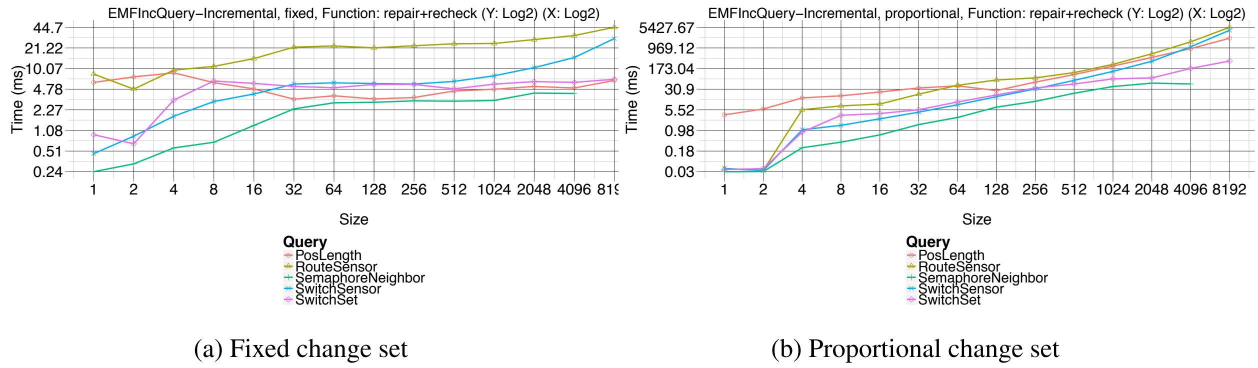


(a) Fixed change set



(b) Proportional change set

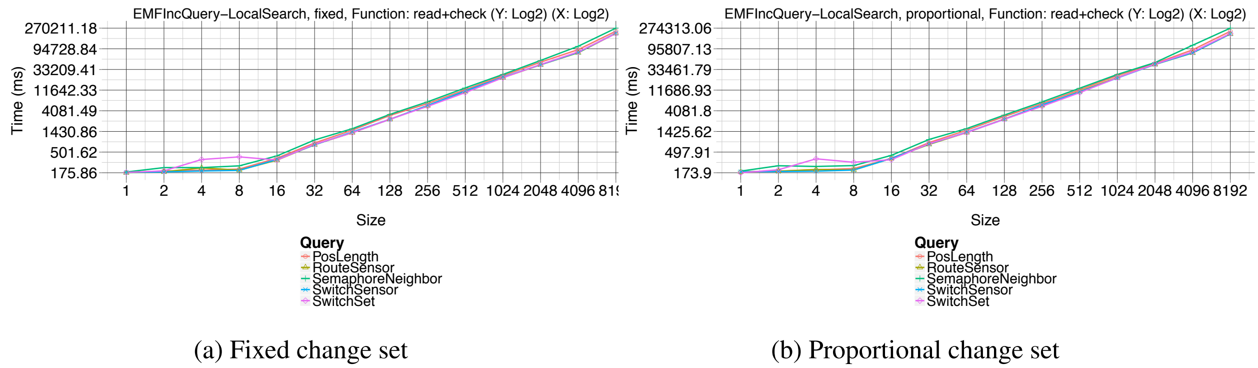
Figure 3: First validation times for the *incremental* query evaluation strategy.



(a) Fixed change set

(b) Proportional change set

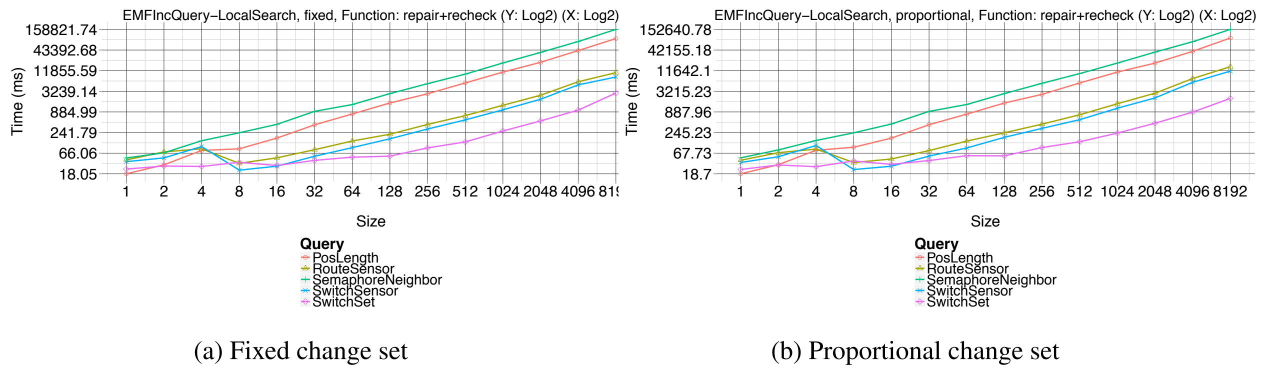
Figure 4: Revalidation times for the *incremental* query evaluation strategy.



(a) Fixed change set

(b) Proportional change set

Figure 5: First validation times for the *local search-based* query evaluation strategy.



(a) Fixed change set

(b) Proportional change set

Figure 6: Revalidation times for the *local search-based* query evaluation strategy.

A.3 Rete Network

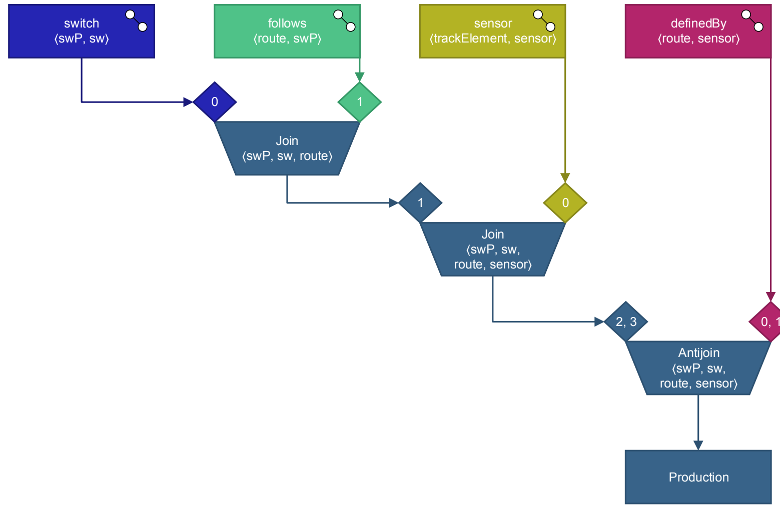


Figure 7: The Rete network for the RouteSensor query.

A.4 Local Search Plan

Local Search Debugger

- extend Route(route)
- extend to target Route.follows(route, switchPosition)
- extend to target SwitchPosition.switch(switchPosition, sw)
- extend to target TrackElement.sensor(sw, sensor)
- NACOperation, pattern: definedBy(route, sensor)
  - check Route.definedBy(route, sensor)
  - Match found
- Match found

routeSensor | definedBy

route	sensor	switchPosition	sw
407	409	415	408
407	447	453	416
407	461	467	454
407	507	513	494
407	553	559	540
673	707	713	700
673	873	879	854

Route 673
follows : SwitchPosition
route : Route
Switch Position 879
switch : Switch
positions : SwitchPosition
Switch 854
sensor : Sensor
elements : TrackElement
Sensor 873

Figure 8: The search plan and the matches for the RouteSensor query.