Evaluation of Optimization Strategies for Incremental Graph Queries*

Gábor Szárnyas, János Maginecz, Dániel Varró
{szarnyas, varro}@mit.bme.hu, janos.maginecz@gmail.com

November 25, 2016

Abstract

The last decade brought considerable improvements in distributed storage and query technologies, known as NoSQL systems. These systems provide quick evaluation of simple retrieval operations and are able to answer certain complex queries in a scalable way, albeit not instantly. Providing scalability and quick response times at the same time for querying large data sets is still a challenging task. Evaluating complex graph queries is particularly difficult, as it requires lots of join, antijoin and filtering operations. This paper presents optimization techniques used in relational database systems and applies them on graph queries. We evaluate various query plans on multiple datasets and discuss the effect of different optimization techniques.

1 Introduction

The key components of Big Data are often defined as *variety*, *velocity* and *volume* [28] of data. Applications operating on continuously changing graphs are a prime example: the semi-structured graph-like nature introduces a *high variety*, changes happen at *high velocity*, and datasets are often *high-volume*. Such applications include fraud detection in financial transactions [27], validation of engineering models [3], and static analysis of source code repositories [35]. These use cases provide a set of complex queries that need to be evaluated continuously on each change of the underlying graph.

Traditional approaches need to reevaluate each query upon each change, which often takes minutes on a large dataset. In contrast, incremental query evaluation caches interim results, hence it only requires reevaluation on a small fragment of the dataset impacted by the change. This leads to significant speedup for large and continuously changing data. Although several approaches exist for incremental query evaluation [9, 20] in the context of expert systems, incremental query evaluation is not in widespread use in graph databases.

In order to predict query performance at runtime, relational databases synthesize and evaluate different query plans which impose a certain ordering on relational algebraic operations prescribed by the query. Optimizing query plans is a challenging task, since a wide variety of query plans may exist even for simple queries with different

*This work was partially supported by the MONDO (EU ICT-611125) project and the MTA-BME Lendület Research Group on Cyber-Physical Systems.

costs. Database engines use heuristics-based optimization techniques and evaluate a cost function for the different query plans [10].

Query plans have been adapted for graph query engines using a local-search based query evaluation strategy where it is called the search plan. Optimization techniques may exploit the type and multiplicity information defined in the graph schema (or metamodel) [29, 22] or rely upon runtime statistics of the instance graph [11, 38, 39].

In case of incremental graph query engines, the structure and the content of caches have the most significant impact on query performance. Therefore, optimization is directed to reduce execution time and memory consumption imposed by a complex network of caches [37].

In this paper, we use heuristic optimization techniques to create different query plans for the incremental evaluation of graph queries describing model validation constraints. We investigate the execution time and scalability of the generated query plans for incremental query evaluation workloads using the open-source Train Benchmark project [33]. The results show that using basic optimization techniques avoiding Cartesian products already results in efficient query plans that scale for models with 9M+ elements, while applying further optimizations (e.g. swapping the operands of join operators) did not have a significant impact.

This paper is an extended version of [17], which evaluates the effect of various optimization techniques on the performance of incremental queries.

Structure of the paper. Section 2 presents the running example, introduces model validation with graph patterns, and demonstrates the usage of graph transformations on models. Section 3 presents labeled graphs, their representation as relations, and the extended relational algebraic operators for processing graphs. Section 4 summarizes the basics of relational graph queries and presents Rete, a widely used incremental graph query algorithm. Section 5 features a set of optimization techniques and shows their application on Rete query networks. Section 6 presents and discusses the results of the performance experiments. Section 7 lists the related optimization approaches, while Section 8 summarizes the paper and outlines future research directions. Appendix A shows the query plan layouts used for the evaluation.

2 Preliminaries

Model-Driven Engineering (MDE) is a widely used technique in many application domains such as automotive, avionics or other cyber-physical systems [40**? ?**]. MDE facilitates the use of models in different phases of design and on various levels of abstraction. These models enable the automated synthesis of various design artifacts (such as source code, configuration files, documentation) and help catch design flaws early by model validation techniques.

In this section, we present the running example and the concepts used in the paper.

2.1 Domain Models for Critical Systems: the Train Benchmark

We use the Train Benchmark [33] as a running example to present both the concepts used in the paper and also as a benchmark framework for evaluating our results. The benchmark defines a domain model for railway networks—the railway domain itself reflects that model validation scenarios are typical in design tools of critical systems.

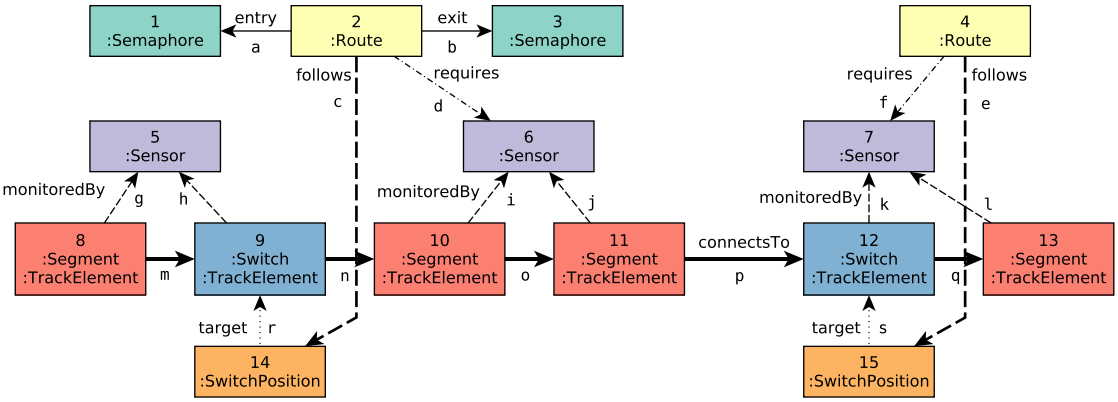

Figure 1: Example railway instance model as a labeled graph.

The railway networks used in the benchmark are composed of typical railroad items, including routes, semaphores, and switches. An example model graph is shown in Figure 1, while the metamodel of the model graph is shown in Figure 2.

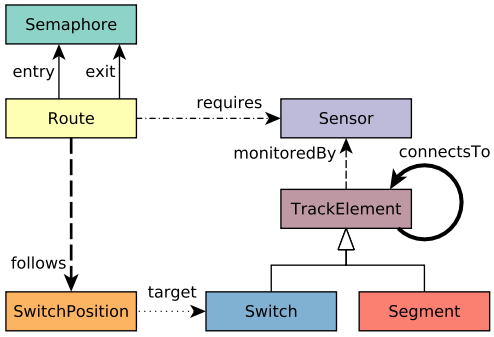

Figure 2: Metamodel of the railway graph.

Formally, structural models (such as this railway network) are often represented as *labeled graphs* (or *typed graphs*), where vertices and edges are annotated with labels (e.g. *Semaphore*, *Route*, *entry*, etc.). Labeled graphs are introduced in Section 3.1.1.

2.2 Model Validation with Graph Queries

Model validation highly depends on repeatedly checking multiple design rules and well-formedness constraints captured in the form of graph patterns [3, 4] over large (graph) models to highlight invalid model elements to systems engineers. A *graph pattern* is a list of symbolic object parameters and the constraints to be satisfied by the parameters. Constraints include positive conditions, negative conditions, and filter conditions. Positive conditions define the structure and labels of the vertices and edges that must be satisfy, while negative conditions define subpatterns which must not be satisfied, while filter conditions express additional constraints (e.g. inequality of vertices). These patterns can be formulated as *graph queries* (or *validation queries*).

A *pattern match* maps each symbolic parameter to a model object, where the mapping satisfies the conditions defined by the constraints. Upon evaluating the validation

query on a model graph, the result is a *match set* that contains all matches for a given pattern. An empty match set for a validation query indicates that the model is well-formed w.r.t. a certain well-formedness constraint, while the matches in a non-empty match set mark the invalid elements that violate the corresponding well-formedness constraint.

The Train Benchmark specifies a set of well-formedness constraints for validating these networks. The benchmark also defines a set of *model transformations*, including *Repair* transformations, which define quick fix-like operations for automatically correcting errors in the model. In total, the benchmark defines six well-formedness constraints with their validation queries and transformations. Here, we only elaborate the two constraints used in the paper. The full set of constraints and transformations of the Train Benchmark is described in [33].

2.2.1 The RouteSensor Constraint

Well-formedness constraint. 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.

Validation query. The RouteSensor query (Figure 3) looks for Sensors (sensor) that are connected to a Route (route) through a Switch (switch) and a SwitchPosition (swP), but the Sensor and the Route are not connected with a requires edge.

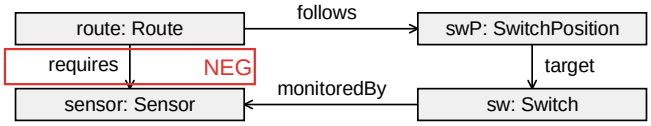

Figure 3: The RouteSensor pattern.

Repair transformation. The missing requires edge is inserted from the route vertex to the sensor vertex in the match (Figure 4), which fixes the violation of the constraint.

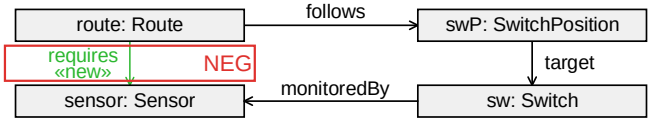

Figure 4: Repair transformation of the RouteSensor pattern.

Example. In the example model (Figure 1), Route 2 violates this constraint, as it is connected to Sensor 5 through SwitchPosition 14 and Switch 9, but the Sensor and the Route are not directly associated. Hence, the validation query return these invalid elements. Running the repair transformation fixes this by inserting a requires edge from Route 2 to Sensor 5.

2.2.2 The SemaphoreNeighbor Constraint

Well-formedness constraint. The SemaphoreNeighbor well-formedness constraint requires Routes which are connected through a pair of Sensors and a pair of TrackElements to belong to the same semaphore.

Validation query. The SemaphoreNeighbor query (Figure 5) checks for Routes (route1) which have an exit Semaphore (semaphore) and a Sensor (sensor1) connected to a TrackElement (te1). This TrackElement is connected to another TrackElement (te2), which is connected to another Sensor (sensor2), which belongs to another, different Route (route2), but the Semaphore is not on the entry of this Route (route2).

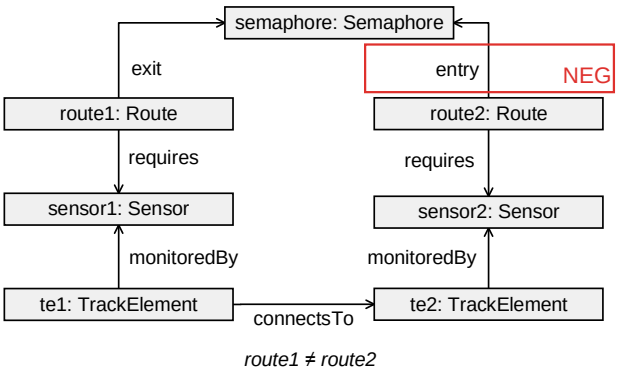

Figure 5: The SemaphoreNeighbor pattern.

Repair transformation. The route2 vertex is connected to the semaphore vertex with an entry edge (Figure 6), which fixes the violation of the constraint.

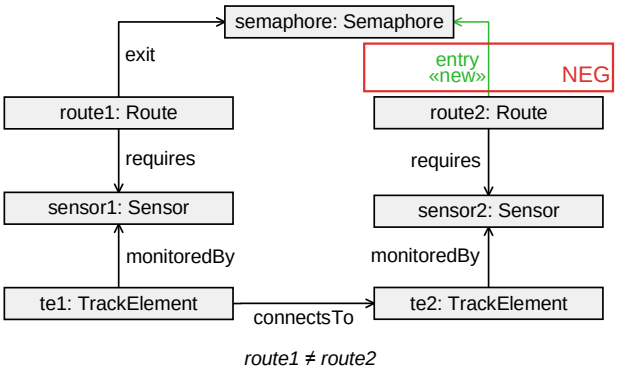

Figure 6: Repair transformation of the SemaphoreNeighbor pattern.

Example. In the example model (Figure 1), Routes 2 and 4 violate this constraint, as they are connected (through Sensors 6, 7 and TrackElements 11, 12), but they do not

belong to the same Semaphore. Hence, the validation query returns these elements as invalids. Running the repair transformation fixes this by inserting an entry edge from Route 4 to Semaphore 3.

3 Relational Query Processing

In this paper, we use the notation of relational algebra for formalizing graph queries. Here, we present the mapping of graphs to relations and present the operators used for defining patterns.

3.1 Relations and Relational Schemas

In relational database theory [10], a *relation* is a subset of the Cartesian product of domains, containing a set of tuples. Each tuple has the same set of *attributes*. The set of *attribute names* for a given relation are called the *relational schema*. Relations are denoted with small letters, while relational schemas are denoted with capital letters, e.g. r is a relation with a corresponding relational schema R.

Note on order of attributes. In the database literature, some authors define a relational schema as a list of attributes [8], while others define it as a set of attributes [10, 18]. In this paper, we define the relational schema as a *set of attributes*, as this choice allows us to formalize queries in a more succinct way.

3.1.1 Labeled Graphs

The notation presented here is based on [14] and [26] with minor adaptations to simplify presentation.

A *labeled graph* is defined as $G = (V, E, \mathsf{src\_trg}, L_v, L_e, l_v, l_e)$, where V is a set of vertices, E is a set of directed edges, $\mathsf{src\_trg} : E \to V \times V$ assigns the source and target vertices to edges.[1] Both vertices and edges in the graph are *labeled* (or *typed*):

- L_v is a set of vertex labels, $l_v : V \to 2^{L_v}$ assigns *a set of labels* to each vertex.
- L_e is a set of edge labels, $l_e : E \to L_e$ assigns *a single label* to each edge.

Property graphs are *labeled graphs* with *attributes* (*properties*) on their vertices and edges. For the sake of brevity, we do not discuss property graphs in this paper and refer the interested reader to [14].

3.1.2 Representing Labeled Graphs as Relations

We map vertices and edges in the graph to tuples in relations. For each vertex label and edge label, we define a separate relation. For the sake of simplicity, we presume that both vertices and edges in the graph have a unique identifier.

[1] While the terms *nodes* and *vertices* are often used as synonyms, we only use the term *vertices* for the graph elements to avoid confusion with *Rete nodes* (introduced in Section 4.1).

Mapping vertices to tuples. Vertices can be trivially mapped to relations of 1-tuples by introducing a relation for each label. The vertices of the example graph in Figure 1 constitute the following relations:

- *Route*(route) = $\{\langle 2\rangle, \langle 4\rangle\}$
- *Segment*(segment) = $\{\langle 8\rangle, \langle 10\rangle, \langle 11\rangle, \langle 13\rangle\}$
- *Semaphore*(semaphore) = $\{\langle 1\rangle, \langle 3\rangle\}$
- *Sensor*(sensor) = $\{\langle 5\rangle, \langle 6\rangle, \langle 7\rangle\}$
- *Switch*(switch) = $\{\langle 9\rangle, \langle 12\rangle\}$
- *SwitchPosition*(switchPosition) = $\{\langle 14\rangle, \langle 15\rangle\}$
- *TrackElement*(trackElement) = $\{\langle 8\rangle, \langle 9\rangle, \langle 10\rangle, \langle 11\rangle, \langle 12\rangle, \langle 13\rangle\}$

Mapping edges to tuples. Each edge is represented by a triple

$$\langle \text{source vertex}, \text{edge}, \text{target vertex}\rangle.$$

The edges of the example graph in Figure 1 constitute the following relations:

- *connectsTo*(trackElement1, connectsTo, trackElement2) = $\{\langle 8, \text{m}, 9\rangle, \langle 9, \text{n}, 10\rangle, \langle 10, \text{o}, 11\rangle, \langle 11, \text{p}, 12\rangle, \langle 12, \text{q}, 13\rangle\}$
- *entry*(route, entry, semaphore) = $\{\langle 2, \text{a}, 1\rangle\}$
- *exit*(route, exit, semaphore) = $\{\langle 2, \text{b}, 3\rangle\}$
- *follows*(route, follows, switchPosition) = $\{\langle 2, \text{c}, 14\rangle, \langle 4, \text{e}, 15\rangle\}$
- *requires*(route, requires, sensor) = $\{\langle 2, \text{d}, 6\rangle, \langle 4, \text{f}, 7\rangle\}$
- *monitoredBy*(trackElement, monitoredBy, sensor) = $\{\langle 8, \text{g}, 5\rangle, \langle 9, \text{h}, 5\rangle, \langle 10, \text{i}, 6\rangle, \langle 11, \text{j}, 6\rangle, \langle 12, \text{k}, 7\rangle, \langle 13, \text{l}, 7\rangle\}$
- *target*(switchPosition, target, switch) = $\{\langle 14, \text{r}, 9\rangle, \langle 15, \text{s}, 12\rangle\}$

Note that in the representation above, attribute names of tuples are only denoted in the relational schemas and not in the tuples of the relation. Without the schema, we cannot determine the *semantics* of tuples, e.g. we cannot decide if the elements in $\langle 8, \text{m}, 9\rangle$ and $\langle 4, \text{f}, 7\rangle$ have the same labels. Hence, we often specify the attribute names for each tuple:

- $\langle \text{trackElement1} : 8, \text{connectsTo} : \text{m}, \text{trackElement2} : 9\rangle$,
- $\langle \text{route} : 4, \text{requires} : \text{f}, \text{sensor} : 7\rangle$.

Using this notation, the *requires* relation can be defined as:

$$\textit{requires} = \{\langle \text{route} : 2, \text{requires} : \text{d}, \text{sensor} : 6\rangle, \langle \text{route} : 4, \text{requires} : \text{f}, \text{sensor} : 7\rangle\}.$$

3.2 Extended Relational Algebra

Relational algebra is a widespread formalism for defining queries on the relational data model. Relational algebra has several extensions, including graph-specific ones [14]. For a detailed discussion of relational algebra operators, the reader is referred to database textbooks [8, 30, 10].

We list basic operators of relational algebra along with graph-specific extensions. We present an example for each operator using the example graph of Figure 1.

3.2.1 Nullary Operators

Get-vertices. The *get-vertices* operator $\bigcirc_{(\mathtt{v}\,:\,\mathsf{t}_1 \wedge \ldots \wedge \mathsf{t}_n)}$ returns a relation of a single attribute v, containing vertices that have *all labels* of $t_1, \ldots, t_n$.

The relations for vertices in Section 3.1.2 can be simply defined using the *get-vertices* operator. For example, the *Route* relation can be expressed as:

$$Route = \bigcirc_{(\mathtt{route}\,:\,\mathsf{Route})}$$

Get-edges. The *get-edges* operator $\Uparrow^{(\mathtt{trg}\,:\,\mathsf{trgLabels})}_{(\mathtt{src}\,:\,\mathsf{srcLabels})}$ [e : edgeLabel] operator returns a relation of three attributes. Each row represents an edge and its vertices: the source vertex `src` (with all labels of srcLabels), the edge `e` (with label `edgeLabel`) and the target vertex `trg` (with all labels of trgLabels).

The relations listed in Section 3.1.2 can be simply defined using the *get-edges* operator. For example the *requires* relation can be expressed as:

$$requires = \Uparrow^{(\mathtt{sensor}\,:\,\mathsf{Sensor})}_{(\mathtt{route}\,:\,\mathsf{Route})} [\mathtt{requires} : \mathsf{requires}]$$

If the edge variable is not used elsewhere in the expression, we often omit its name and use the operator as $\Uparrow^{(\mathtt{trg}\,:\,\mathsf{trgLabels})}_{(\mathtt{src}\,:\,\mathsf{srcLabels})}$ [: edgeLabel].

3.2.2 Unary Operators

Projection. The *projection* operator π reduces the dimension of the relation by only keeping a specific set of attributes: $t = \pi_{\mathsf{A}_1,\ldots,\mathsf{A}_n}(r)$.

For example, the following expression returns the route vertices from the *requires* relation.

$$\pi_{\mathsf{route}}(requires) = \{\langle \mathsf{route} : 2\rangle, \langle \mathsf{route} : 4\rangle\}$$

Selection. The *selection* operator σ filters the relation according to some criteria: $t = \sigma_\theta(r)$, where predicate θ is a propositional formula. The operator selects all tuples in r for which θ holds.

For example, the following expression returns edges from the requires relation, which start from route 2.

$$\sigma_{\mathsf{route}=2}(requires) = \{\langle \mathsf{route} : 2, \mathsf{requires} : \mathrm{d}, \mathsf{sensor} : 6\rangle\}$$

3.2.3 Binary Operators

Cartesian Product. The $\times$ operator produces the *Cartesian product*

$$t = r \times s,$$

where t holds all tuples that are a union of exactly one tuple from r and exactly one tuple from s.

For example, the following expression produces the Cartesian product of the *follows* and *requires* relations:

$$
\begin{aligned}
&\mathit{follows} \times \mathit{requires} = \{ \\
&\quad \langle \mathsf{route}_1 : 2, \mathsf{follows} : \mathrm{c}, \mathsf{switchPosition} : 14, \mathsf{route}_2 : 2, \mathsf{requires} : \mathrm{d}, \mathsf{sensor} : 6 \rangle, \\
&\quad \langle \mathsf{route}_1 : 2, \mathsf{follows} : \mathrm{c}, \mathsf{switchPosition} : 14, \mathsf{route}_2 : 4, \mathsf{requires} : \mathrm{f}, \mathsf{sensor} : 7 \rangle, \\
&\quad \langle \mathsf{route}_1 : 4, \mathsf{follows} : \mathrm{e}, \mathsf{switchPosition} : 15, \mathsf{route}_2 : 2, \mathsf{requires} : \mathrm{d}, \mathsf{sensor} : 6 \rangle, \\
&\quad \langle \mathsf{route}_1 : 4, \mathsf{follows} : \mathrm{e}, \mathsf{switchPosition} : 15, \mathsf{route}_2 : 4, \mathsf{requires} : \mathrm{f}, \mathsf{sensor} : 7 \rangle \\
&\}
\end{aligned}
$$

Natural Join. The result of the *natural join* operator $\bowtie$ is determined by creating the Cartesian product of the relations, then filtering those tuples which are equal on the attributes that share a common name. The combined tuples are projected: from the attributes present in both of the two input relations, we only keep a single one. Thus, the join operator is defined as:

$$r \bowtie s = \pi_{R \cup S} \left(\sigma_{\mathsf{r.A}_1 = \mathsf{s.A}_1 \wedge \ldots \wedge \mathsf{r.A}_n = \mathsf{s.A}_n)} \left(r \times s \right) \right),$$

where R and S are the relational schemas of relations r and s, respectively; $\{A_1, \ldots, A_n\}$ is the set of attributes that occur both in schemas R and S, i.e. $R \cap S = \{A_1, \ldots, A_n\}$. Note that if the set of common attributes is empty, the *natural join* operator is equivalent to the Cartesian product of the relations. The join operator is both commutative and associative: $r \bowtie s = s \bowtie r$ and $(r \bowtie s) \bowtie t = r \bowtie (s \bowtie t)$.

The join operator can connect vertices, edges and subgraphs to each other. For example, subgraphs of three vertices and two edges along the follows and requires edges can be queried as:

$$
\begin{aligned}
&\mathit{follows} \bowtie \mathit{requires} = \{ \\
&\quad \langle \mathsf{route} : 2, \mathsf{follows} : \mathrm{c}, \mathsf{switchPosition} : 14, \mathsf{requires} : \mathrm{d}, \mathsf{sensor} : 6 \rangle, \\
&\quad \langle \mathsf{route} : 4, \mathsf{follows} : \mathrm{e}, \mathsf{switchPosition} : 15, \mathsf{requires} : \mathrm{f}, \mathsf{sensor} : 7 \rangle \\
&\}
\end{aligned}
$$

Antijoin. The *antijoin* operator $\triangleright$ (also known as *left anti semijoin*) collects the tuples from the left relation r which have no matching pair in the right relation s:

$$t = r \triangleright s = r \setminus \pi_R \left(r \bowtie s \right),$$

where π_R denotes a projection operator, which only keeps the attributes of the schema over relation r. The definition can be also formulated using the set of common attributes $R \cap S$:

$$t = r \setminus \pi_R \left(r \bowtie s \right) = r \setminus \left(r \bowtie \pi_{R \cap S}(s) \right)$$

Unlike the natural join operator, the antijoin operator is not commutative and not associative.

The antijoin operator can express negative conditions. For example, the triples for follows edges that do not have an exit edge on their route can be queried as:

$$\mathit{follows} \triangleright \mathit{exit} = \{ \langle \mathsf{route} : 4, \mathsf{follows} : \mathrm{e}, \mathsf{switchPosition} : 15 \rangle \}$$

Note on operator precedence. The join and antijoin operators have the same precedence, i.e. if the order between join and antijoin operations is not indicated, the query is processed from left to right.

3.3 Graph Patterns in Relational Algebra

The graph patterns (such as the validation queries introduced in Section 2.2) can be formalized using relational algebra: edges of the pattern are selected with the *get-edges* operator, connections are enforced using the *natural join* operator, negative subpatterns are translated using *antijoin*, and filtering is captured by the *selection* operator. Following this definition, we give relational algebra expressions for both validation queries.

3.3.1 Query RouteSensor

The RouteSensor query can be formalized as:

$$\Uparrow^{(\texttt{swP}:\ \text{SwitchPosition})}_{(\texttt{route}:\ \text{Route})}[:\ \text{follows}] \bowtie \Uparrow^{(\texttt{sw}:\ \text{Switch})}_{(\texttt{swP}:\ \text{SwitchPosition})}[:\ \text{target}] \bowtie$$
$$\Uparrow^{(\texttt{sensor}:\ \text{Sensor})}_{(\texttt{sw}:\ \text{Switch})}[:\ \text{monitoredBy}] \rhd \Uparrow^{(\texttt{sensor}:\ \text{Sensor})}_{(\texttt{route}:\ \text{Route})}[:\ \text{requires}]$$

This query is of medium complexity, using 4 get-edges, 2 natural join and 1 antijoin operators.

3.3.2 Query SemaphoreNeighbor

The SemaphoreNeighbor query can be formalized as:

$$\sigma_{\texttt{route1}\neq\texttt{route2}}\Big(\Uparrow^{(\texttt{te2}:\ \text{TrackElement})}_{(\texttt{te1}:\ \text{TrackElement})}[:\ \text{connectsTo}] \bowtie$$
$$\Uparrow^{(\texttt{sensor1}:\ \text{Sensor})}_{(\texttt{te1}:\ \text{TrackElement})}[:\ \text{monitoredBy}] \bowtie \Uparrow^{(\texttt{sensor2}:\ \text{Sensor})}_{(\texttt{te2}:\ \text{TrackElement})}[:\ \text{monitoredBy}] \bowtie$$
$$\Uparrow^{(\texttt{semaphore}:\ \text{Semaphore})}_{(\texttt{route1}:\ \text{Route})}[:\ \text{exit}] \bowtie \Uparrow^{(\texttt{sensor1}:\ \text{Sensor})}_{(\texttt{route1}:\ \text{Route})}[:\ \text{requires}] \bowtie$$
$$\Uparrow^{(\texttt{sensor2}:\ \text{Sensor})}_{(\texttt{route2}:\ \text{Route})}[:\ \text{requires}]\Big) \rhd \Uparrow^{(\texttt{semaphore}:\ \text{Semaphore})}_{(\texttt{route2}:\ \text{Route})}[:\ \text{entry}]$$

This query is significantly more complex, using 7 get-edges, 5 natural join, 1 antijoin and 1 selection operators.

4 Incremental Query Evaluation

In many use cases, queries are continuously evaluated, while the data only changes rarely and to a small degree. The validation queries in MDE are a typical example of such a workload. The goal of *incremental query evaluation* is to speed up such queries, using the (partial) results obtained during the previous executions of the query and only computing the effect of the latest set of changes.

Incremental query evaluation algorithms typically use additional data structures for caching interim results. This implies that they usually consume more memory than non-incremental, search-based algorithms. In other words, they trade memory consumption for execution speed. This approach, called *space–time tradeoff*, is well-known and widely used in computer science [13].

Numerous algorithms were proposed for incremental pattern matching. Mostly, these algorithms originate from the field of rule-based expert systems. In this paper, we use the *Rete algorithm* [9], which creates a *data flow network* for evaluating relational queries.

4.1 Overview of Rete Networks

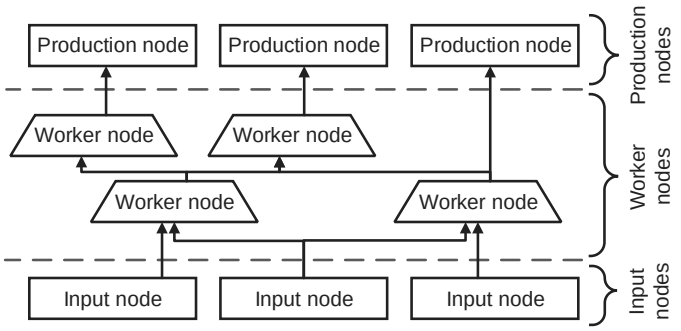

Figure 7: The structure of the Rete propagation network.

The Rete algorithm constructs a network of three types of nodes. Following Figure 7, from bottom to top:

1. *Input nodes* are responsible for indexing the graph, i.e. they store the appropriate tuples for vertices and edges in the graph. They are also responsible for sending *change sets* as *update messages* to worker nodes that are *subscribed* to them.

2. *Worker nodes* perform a relational algebraic operation on their input and propagate the results to other worker nodes or production nodes. Some worker nodes are *stateful*: they store partial query results in their memory to allow incremental reevaluation. Worker nodes have two types: *unary nodes* have a single input slot, *binary nodes* have two input slots.

3. *Production nodes* are terminators that provide an interface for fetching the results.

The Rete network operates as follows. First, the network computes the set of pattern matches in the graph. Then upon a change in the graph, the network is incrementally maintained by propagating *update messages* (also known as *deltas*, denoted with the Δ character). Adding new graph matches to the result set is expressed as *positive update messages*, while removing matches results in *negative update messages*.

In the following, we discuss Rete nodes in detail. For unary and binary nodes, we formulate the *maintenance operations*, which are performed upon receiving an update message. For these operations, we denote the output relation by t, the updated output relation by t', and the propagated update message on the output by Δt. If the *propagated update message* is a positive update, $t' = t \cup \Delta t$, if it is a negative update, $t' = t \setminus \Delta t$.

4.2 Input Nodes

Input nodes provide the relations for each label of the graph. For example, the input node for the requires edge label of example the graph (Figure 1) returns tuples that are

currently in the *requires* relation: $\{\langle 2, \mathrm{d}, 6\rangle, \langle 4, \mathrm{f}, 7\rangle\}$. This input node is also responsible for propagating changes to worker nodes in the network:

- If a requires edge 't' is inserted from vertex 2 to 5, the input node sends a positive update message to its subscriber nodes with the change set $\{\langle 2, \mathrm{t}, 4\rangle\}$.
- If the edge 'd' between vertices 2 and 6 is deleted, the input node sends a negative update to its subscriber nodes with the change set $\{\langle 2, \mathrm{d}, 6\rangle\}$.

The relations contained by input nodes can be defined with nullary operators (Section 3.2.1): input nodes indexing vertices implement the *get-vertices* operator, while input nodes indexing edges implement the *get-edges* operator.

4.3 Unary Nodes

Unary nodes have one input slot. They filter or transform the tuples of the parent node according to certain criteria. In the following, the relation representing the input tuples is denoted with r, the relation representing the output tuples is denoted with t, and the operator processing the input is denoted with α:

$$t = \alpha(r).$$

Maintenance. In the following, we assume that the α operator is *distributive* w.r.t. the union ($\cup$) and set minus ($\setminus$) operators. If a unary node receives an update Δr, it performs the operation and computes the change set. For *positive updates*, the result (t') and the changeset (Δt) are:

$$\begin{aligned} t' &\equiv \alpha(r \cup \Delta r) \\ &= \alpha(r) \cup \alpha(\Delta r) \\ &= t \cup \underbrace{\alpha(\Delta r)}_{\Delta t} \end{aligned}$$

Similarly, for *negative updates*:

$$\begin{aligned} t' &\equiv \alpha(r \setminus \Delta r) \\ &= \alpha(r) \setminus \alpha(\Delta r) \\ &= t \setminus \underbrace{\alpha(\Delta r)}_{\Delta t} \end{aligned}$$

Unary nodes are often implemented as *stateless* nodes, i.e. they do not store the results of the previous executions. Instead, these results are cached in their subscribers, e.g. indexers of *binary nodes* (Section 4.4) or *production nodes* (Section 4.5).

As their name suggests, unary nodes implement unary relational algebraic operators (Section 3.2.2):

- The *projection node* performs a projection operation on the input relation.
- The *selection node* performs a selection operation on the input relation.

As both the projection and the selection operators are distributive w.r.t. the union and set minus operators, their results can be maintained by performing the operation for the change set Δr.

4.4 Binary Nodes

Binary nodes have two input slots: the *primary* (p) and the *secondary* (s). Binary node implementations typically cache both their input relations in *indexers*.

4.4.1 Natural Join Node

Maintenance. In the following, we define the maintenance operations for natural join nodes. If a natural join node receives a *positive update* Δp on its *primary* input slot, the result (t') and the change set (Δt) are determined as follows:

$$\begin{aligned} t' &\equiv (p \cup \Delta p) \bowtie s \\ &= (p \bowtie s) \cup (\Delta p \bowtie s) \\ &= t \cup \underbrace{(\Delta p \bowtie s)}_{\Delta t} \end{aligned}$$

If the node receives a *positive update* Δs on its *secondary* input slot, the result (t') and the change set (Δt) are the following:

$$\begin{aligned} t' &\equiv p \bowtie (s \cup \Delta s) \\ &= (p \bowtie s) \cup (p \bowtie \Delta s) \\ &= t \cup \underbrace{(p \bowtie \Delta s)}_{\Delta t} \end{aligned}$$

For *negative updates*, the changeset is the same, but it is propagated as a *negative update*. The result is $t' = t \setminus (\Delta p \bowtie s)$ and $t' = t \setminus (p \bowtie \Delta s)$, for updates messages on the primary and the secondary input slots, respectively.

4.4.2 Antijoin Node

Maintenance. As the antijoin operator is not commutative, handling update messages requires us to distinguish between the following cases:

- Update on the primary slot.
  - Positive update: send a *positive update* for each incoming tuple for which there is no match on the secondary indexer.

$$\begin{aligned} t' &\equiv (p \cup \Delta p) \rhd s \\ &= (p \rhd s) \cup (\Delta p \rhd s) \\ &= t \cup \underbrace{(\Delta p \rhd s)}_{\Delta t} \end{aligned}$$

  - Negative update: send a *negative update* with the following tuples:

$$\Delta t = \Delta p \rhd s$$

- Update on the secondary slot. This case is more difficult to handle, so we recall the definition of the antijoin operator from Section 3.2.3 for relations p and s:

$$t \equiv p \rhd s = p \setminus \left(p \bowtie \pi_{P \cap S}(s)\right),$$

– For positive updates, the result set can be expressed as:

$$\begin{aligned} t' &\equiv p \rhd (s \cup \Delta s) \\ &= p \setminus \left(p \bowtie \pi_{P \cap S}(s \cup \Delta s)\right) \end{aligned}$$

Positive updates on the secondary indexer result in *negative updates* on the result set, so that $t' = t \setminus \Delta t$, hence $\Delta t = t \setminus t'$.
For sets $A, B \subseteq C$, the following equality holds: $(C \setminus A) \setminus (C \setminus B) = B \setminus A$. Applying this with $C = p$ and using the distributive property of the natural join operator, the change set can be determined as:

$$\begin{aligned} \Delta t = t \setminus t' &= \overbrace{\left[p \setminus \left(p \bowtie \pi_{P \cap S}(s)\right)\right]}^{t} \setminus \overbrace{\left[p \setminus \left(p \bowtie \pi_{P \cap S}(s \cup \Delta s)\right)\right]}^{t'} \\ &= \left(p \bowtie \pi_{P \cap S}(s \cup \Delta s)\right) \setminus \left(p \bowtie \pi_{P \cap S}(s)\right) \\ &= p \bowtie \pi_{P \cap S}(\Delta s) \end{aligned}$$

This implies that we send a *negative update* for each tuple in the primary indexer, which have a match in the incoming tuples.

– For negative updates, the result set can be expressed as:

$$\begin{aligned} t' &\equiv p \rhd (s \setminus \Delta s) \\ &= p \setminus \left(p \bowtie \pi_{P \cap S}(s \setminus \Delta s)\right), \end{aligned}$$

Negative updates may result in *positive updates* on the result set. Since $t' = t \cup \Delta t$, we can define $\Delta t = t' \setminus t$:

$$\begin{aligned} \Delta t = t' \setminus t &= \overbrace{\left[p \setminus \left(p \bowtie \pi_{P \cap S}(s \setminus \Delta s)\right)\right]}^{t'} \setminus \overbrace{\left[p \setminus \left(p \bowtie \pi_{P \cap S}(s)\right)\right]}^{t} \\ &= \underbrace{\left(p \bowtie \pi_{P \cap S}(s)\right)}_{x} \setminus \underbrace{\left(p \bowtie \pi_{P \cap S}(s \setminus \Delta s)\right)}_{y} \end{aligned}$$

Although this change set may seem difficult to calculate, we point out that both x and y can be maintained incrementally. Furthermore, they only grow linearly in the size of p, as the join operator does not introduce new attributes, hence it can only reduce the number of elements in the relation.

4.5 Production Nodes

Production nodes are terminators that provide an interface for fetching results of a query (the match set) and also propagate the changes introduced by the latest update message.[2]

Maintenance. The change set is defined as:

$$\Delta t \equiv \bigcup_{i=1}^{n} \Delta r_i,$$

where $\Delta r_1, \Delta r_2, \ldots, \Delta r_n$ are the update messages triggered by the last change.

[2] In popular Rete implementations, client are usually subscribed to the production nodes and notified about the changes in the result set.

5 Rete Network Optimization

This section provides an overview of the query optimization process and discusses optimization techniques for Rete networks.

5.1 Relational Query Optimization

5.1.1 Workflow

Figure 8 shows the generic workflow of relational query optimization [10]. Users and developers typically formulate their *query specification* in a high-level declarative language (such as SQL for relational databases, SPARQL for semantic databases [42] or Cypher for graph databases [34]). The query is interpreted by a *query parser* and compiled to a *query plan*, consisting of *input relations* and *relational algebraic operations*.

Modern database management systems use a sophisticated *query optimizer* module, which generates an efficient query plan for evaluating the query. The *query engine* uses the query plan for evaluating the query on the database.

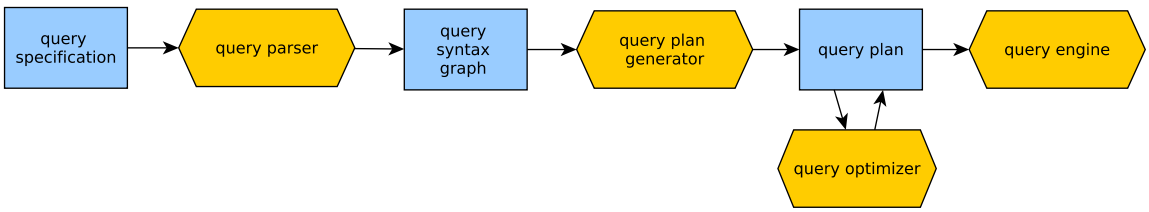

Figure 8: Generic workflow of query optimization in databases.

5.1.2 Optimization Techniques

Cost-based optimization. Cost-based query optimizers typically use a *cost function* for estimating the resources required for executing the query (e.g. execution time, memory consumption, number of disk operations). The goal of the optimization process is to minimize the cost function. Finding the best query plan for a certain query is a computationally expensive task. In particular, exhaustive search is often infeasible in practice. For example, due to the commutative and associative properties of the join operation, the join of n relations can be formalized as $\frac{(2(n-1))!}{(n-1)!}$ different expressions [30], all of which produce the same results, but may vary largely in performance.

Heuristic optimization. Instead of implementing sophisticated cost estimation algorithms, optimizers often use simple heuristics, e.g. they push selections closer to the leaves of the relational algebra tree and reorder joins using their associativity.

5.2 Optimization of Rete Networks

A relational algebraic *query plan* can be trivially transformed to a *Rete network* by instantiating the appropriate Rete nodes. Nullary operators are transformed to input nodes. Unary and binary operators are transformed to worker nodes. For fetching the results, a *production node* is inserted to the root of the tree.

However, simply transforming a "well-optimized" query plan by itself does not guarantee that the Rete network have good performance. For Rete networks, one of the most important optimization goals is to minimize the *query execution time*. In general, optimization of Rete network layouts has three key subgoals:

1. Reduce the size of the Rete network, i.e. the number of Rete nodes [37].
2. Reduce communication between Rete nodes [19], i.e. the amount of update messages required for evaluating (and reevaluating) the query.
3. Reduce the number of tuples stored in Rete nodes, i.e. the total memory consumption of the Rete network.

These goals are interdependent: optimizing along one goal is often beneficial for other goals as well.

5.3 Optimization of the Train Benchmark Validation Queries

To investigate the effect of the layout of the query plan on query performance, we took the queries introduced in Section 2.2 and manually generated different query plans using heuristic optimization techniques: reordering join/antijoin operators or pushing selection operators close to the leaves or root of the tree. We derived 3 different query plans for RouteSensor and 6 query plans for SemaphoreNeighbor. For each query plan, the corresponding Rete network layout is presented in Appendix A (Figure 12–20).

5.3.1 Optimization of Query RouteSensor

We designed three query plans for the RouteSensor query. Figures 12–14 show the layouts for these plans.

$$\underline{A:}\quad \Uparrow^{(\texttt{swP}:\,\text{SwitchPosition})}_{(\texttt{route}:\,\text{Route})}[:\,\text{follows}] \bowtie \Uparrow^{(\texttt{sw}:\,\text{Switch})}_{(\texttt{swP}:\,\text{SwitchPosition})}[:\,\text{target}] \bowtie \Uparrow^{(\texttt{sensor}:\,\text{Sensor})}_{(\texttt{sw}:\,\text{Switch})}[:\,\text{monitoredBy}] \triangleright \Uparrow^{(\texttt{sensor}:\,\text{Sensor})}_{(\texttt{route}:\,\text{Route})}[:\,\text{requires}]$$

$$\underline{B:}\quad \Uparrow^{(\texttt{sw}:\,\text{Switch})}_{(\texttt{swP}:\,\text{SwitchPosition})}[:\,\text{target}] \bowtie \Uparrow^{(\texttt{sensor}:\,\text{Sensor})}_{(\texttt{sw}:\,\text{Switch})}[:\,\text{monitoredBy}] \bowtie \Uparrow^{(\texttt{swP}:\,\text{SwitchPosition})}_{(\texttt{route}:\,\text{Route})}[:\,\text{follows}] \triangleright \Uparrow^{(\texttt{sensor}:\,\text{Sensor})}_{(\texttt{route}:\,\text{Route})}[:\,\text{requires}]$$

$$\underline{C:}\quad \Uparrow^{(\texttt{swP}:\,\text{SwitchPosition})}_{(\texttt{route}:\,\text{Route})}[:\,\text{follows}] \bowtie \Uparrow^{(\texttt{sensor}:\,\text{Sensor})}_{(\texttt{sw}:\,\text{Switch})}[:\,\text{monitoredBy}] \triangleright \Uparrow^{(\texttt{sensor}:\,\text{Sensor})}_{(\texttt{route}:\,\text{Route})}[:\,\text{requires}] \bowtie \Uparrow^{(\texttt{sw}:\,\text{Switch})}_{(\texttt{swP}:\,\text{SwitchPosition})}[:\,\text{target}]$$

Variants *A* and *B* only differ in the order of the join operations. However, variant *C* uses a join with without common attributes, which results in a Cartesian product operation.

5.3.2 Optimization of Query SemaphoreNeighbor

As the SemaphoreNeighbor query is more complex than RouteSensor, it allows more sophisticated optimization methods. We designed six query plans:

- Variants *A*–*D* use different orders for the join/antijoin operators.

- Similarly to query RouteSensor, there are query plans requiring a Cartesian product: both E and F perform a join on relations without common attributes.

Figures 15–20 show the layouts for these plans.

$$\underline{A:}\quad \sigma_{\texttt{route1}\neq\texttt{route2}}\Big(\Uparrow^{(\texttt{te2}:\,\text{TrackElement})}_{(\texttt{te1}:\,\text{TrackElement})}[:\text{connectsTo}]\bowtie \Uparrow^{(\texttt{sensor1}:\,\text{Sensor})}_{(\texttt{te1}:\,\text{TrackElement})}[:\text{monitoredBy}]\bowtie\Uparrow^{(\texttt{sensor2}:\,\text{Sensor})}_{(\texttt{te2}:\,\text{TrackElement})}[:\text{monitoredBy}]\bowtie \Uparrow^{(\texttt{semaphore}:\,\text{Semaphore})}_{(\texttt{route1}:\,\text{Route})}[:\text{exit}]\bowtie\Uparrow^{(\texttt{sensor1}:\,\text{Sensor})}_{(\texttt{route1}:\,\text{Route})}[:\text{requires}]\bowtie \Uparrow^{(\texttt{sensor2}:\,\text{Sensor})}_{(\texttt{route2}:\,\text{Route})}[:\text{requires}]\Big)\rhd\ \Uparrow^{(\texttt{semaphore}:\,\text{Semaphore})}_{(\texttt{route2}:\,\text{Route})}[:\text{entry}]$$

$$\underline{B:}\quad \sigma_{\texttt{route1}\neq\texttt{route2}}\Big(\Uparrow^{(\texttt{te2}:\,\text{TrackElement})}_{(\texttt{te1}:\,\text{TrackElement})}[:\text{connectsTo}]\bowtie \Uparrow^{(\texttt{sensor1}:\,\text{Sensor})}_{(\texttt{te1}:\,\text{TrackElement})}[:\text{monitoredBy}]\bowtie\Uparrow^{(\texttt{sensor1}:\,\text{Sensor})}_{(\texttt{route1}:\,\text{Route})}[:\text{requires}]\bowtie \Uparrow^{(\texttt{sensor2}:\,\text{Sensor})}_{(\texttt{te2}:\,\text{TrackElement})}[:\text{monitoredBy}]\bowtie\Uparrow^{(\texttt{sensor2}:\,\text{Sensor})}_{(\texttt{route2}:\,\text{Route})}[:\text{requires}]\Big)\bowtie \Uparrow^{(\texttt{semaphore}:\,\text{Semaphore})}_{(\texttt{route1}:\,\text{Route})}[:\text{exit}]\rhd\ \Uparrow^{(\texttt{semaphore}:\,\text{Semaphore})}_{(\texttt{route2}:\,\text{Route})}[:\text{entry}]$$

$$\underline{C:}\quad \sigma_{\texttt{route1}\neq\texttt{route2}}\Big(\Uparrow^{(\texttt{te2}:\,\text{TrackElement})}_{(\texttt{te1}:\,\text{TrackElement})}[:\text{connectsTo}]\bowtie \Uparrow^{(\texttt{sensor1}:\,\text{Sensor})}_{(\texttt{te1}:\,\text{TrackElement})}[:\text{monitoredBy}]\bowtie\Uparrow^{(\texttt{sensor2}:\,\text{Sensor})}_{(\texttt{te2}:\,\text{TrackElement})}[:\text{monitoredBy}]\bowtie \Uparrow^{(\texttt{sensor1}:\,\text{Sensor})}_{(\texttt{route1}:\,\text{Route})}[:\text{requires}]\bowtie\Uparrow^{(\texttt{semaphore}:\,\text{Semaphore})}_{(\texttt{route1}:\,\text{Route})}[:\text{exit}]\bowtie \Uparrow^{(\texttt{sensor2}:\,\text{Sensor})}_{(\texttt{route2}:\,\text{Route})}[:\text{requires}]\Big)\rhd\ \Uparrow^{(\texttt{semaphore}:\,\text{Semaphore})}_{(\texttt{route2}:\,\text{Route})}[:\text{entry}]$$

$$\underline{D:}\quad \sigma_{\texttt{route1}\neq\texttt{route2}}\Big(\Uparrow^{(\texttt{te2}:\,\text{TrackElement})}_{(\texttt{te1}:\,\text{TrackElement})}[:\text{connectsTo}]\bowtie \Uparrow^{(\texttt{semaphore}:\,\text{Semaphore})}_{(\texttt{route1}:\,\text{Route})}[:\text{exit}]\bowtie\Uparrow^{(\texttt{sensor1}:\,\text{Sensor})}_{(\texttt{route1}:\,\text{Route})}[:\text{requires}]\bowtie \Uparrow^{(\texttt{sensor1}:\,\text{Sensor})}_{(\texttt{te1}:\,\text{TrackElement})}[:\text{monitoredBy}]\bowtie\Uparrow^{(\texttt{sensor2}:\,\text{Sensor})}_{(\texttt{te2}:\,\text{TrackElement})}[:\text{monitoredBy}]\bowtie \Uparrow^{(\texttt{sensor2}:\,\text{Sensor})}_{(\texttt{route2}:\,\text{Route})}[:\text{requires}]\Big)\rhd\ \Uparrow^{(\texttt{semaphore}:\,\text{Semaphore})}_{(\texttt{route2}:\,\text{Route})}[:\text{entry}]$$

$$\underline{E:}\quad \sigma_{\texttt{route1}\neq\texttt{route2}}\Big(\Uparrow^{(\texttt{semaphore}:\,\text{Semaphore})}_{(\texttt{route1}:\,\text{Route})}[:\text{exit}]\bowtie\Uparrow^{(\texttt{sensor2}:\,\text{Sensor})}_{(\texttt{route2}:\,\text{Route})}[:\text{requires}]\Big)\rhd \Uparrow^{(\texttt{semaphore}:\,\text{Semaphore})}_{(\texttt{route2}:\,\text{Route})}[:\text{entry}]\bowtie\Uparrow^{(\texttt{sensor1}:\,\text{Sensor})}_{(\texttt{route1}:\,\text{Route})}[:\text{requires}]\bowtie \Uparrow^{(\texttt{sensor1}:\,\text{Sensor})}_{(\texttt{te1}:\,\text{TrackElement})}[:\text{monitoredBy}]\bowtie\Uparrow^{(\texttt{te2}:\,\text{TrackElement})}_{(\texttt{te1}:\,\text{TrackElement})}[:\text{connectsTo}]\bowtie \Uparrow^{(\texttt{sensor2}:\,\text{Sensor})}_{(\texttt{te2}:\,\text{TrackElement})}[:\text{monitoredBy}]$$

$$\underline{F:}\quad \sigma_{\texttt{route1}\neq\texttt{route2}}\Big(\Uparrow^{(\texttt{sensor1}:\,\text{Sensor})}_{(\texttt{te1}:\,\text{TrackElement})}[:\text{monitoredBy}]\bowtie \Uparrow^{(\texttt{sensor2}:\,\text{Sensor})}_{(\texttt{te2}:\,\text{TrackElement})}[:\text{monitoredBy}]\bowtie\Uparrow^{(\texttt{sensor1}:\,\text{Sensor})}_{(\texttt{route1}:\,\text{Route})}[:\text{requires}]\bowtie \Uparrow^{(\texttt{sensor2}:\,\text{Sensor})}_{(\texttt{route2}:\,\text{Route})}[:\text{requires}]\Big)\bowtie\Uparrow^{(\texttt{semaphore}:\,\text{Semaphore})}_{(\texttt{route1}:\,\text{Route})}[:\text{exit}]\bowtie \Uparrow^{(\texttt{te2}:\,\text{TrackElement})}_{(\texttt{te1}:\,\text{TrackElement})}[:\text{connectsTo}]\rhd\ \Uparrow^{(\texttt{semaphore}:\,\text{Semaphore})}_{(\texttt{route2}:\,\text{Route})}[:\text{entry}]$$

6 Evaluation

We performed a series of measurements to assess the performance of various Rete layouts for evaluating a query and a corresponding transformation.

6.1 Benchmark Setup

For the measurements, we used the open-source Train Benchmark [33] framework (which also provided the example in Section 2.1).

Goal. The goal of our experiments was to measure the *execution time* and *scalablity* of incremental query evaluation. This alignes with the goals originally defined in the Train Benchmark, but instead of comparing different tools, our goal is to compare the performance of different query plans.

Framework. The Train Benchmark provides an extensible framework to allow implementations of the benchmark for different tools. The framework features end-to-end automation, including building the source code, generating instance models, running benchmarks, and visualizing results. For the measurements in this paper, we implemented an extension in the framework that allowed us to use different query plants for the same query.

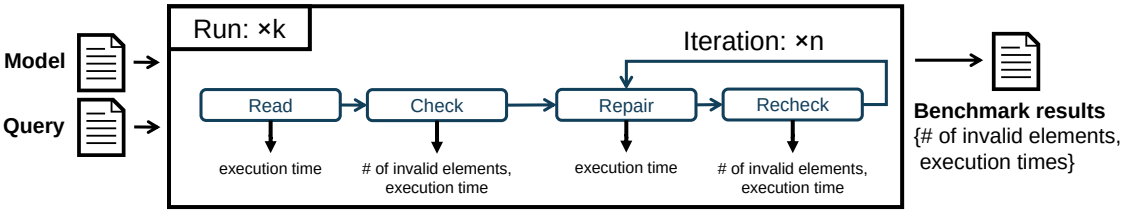

Figure 9: Phases of the *Repair* scenario.

Methodology. The measurements were performed according to the workflow of the *Repair* scenario, shown in Figure 9. In this scenario, the model is loaded and validated (Read and Check phases). Next, a subset of the model is transformed and revalidated (Repair and Recheck phases). This aims to simulate a workload similar to a user applying quick fixes to the model.

During the benchmark, each measurement was executed 10 times ($k = 10$) in a separate Java Virtual Machine (instantiated for each measurement), with 100 iterations for the Repair–Recheck phases ($n = 100$). Each measurement had to complete within a timout limit of 15 minutes, else its process was terminated and its results were discarded.

Environment. The benchmark was performed on a virtual machine with an eight-core, 2.4 GHz Intel Xeon E5-2650L CPU with 16 GBs of RAM, and an SSD hard drive. The machine was running a 64-bit Ubuntu 16.04 server operating system and the Oracle JDK 1.8.0_111 with 12 GBs of heap memory. The Rete-based query engine is implemented in Scala 2.11.[3]

[3]The implementation is available as an open-source project at `https://github.com/FTSRG/ingraph`.

6.2 Results Analysis

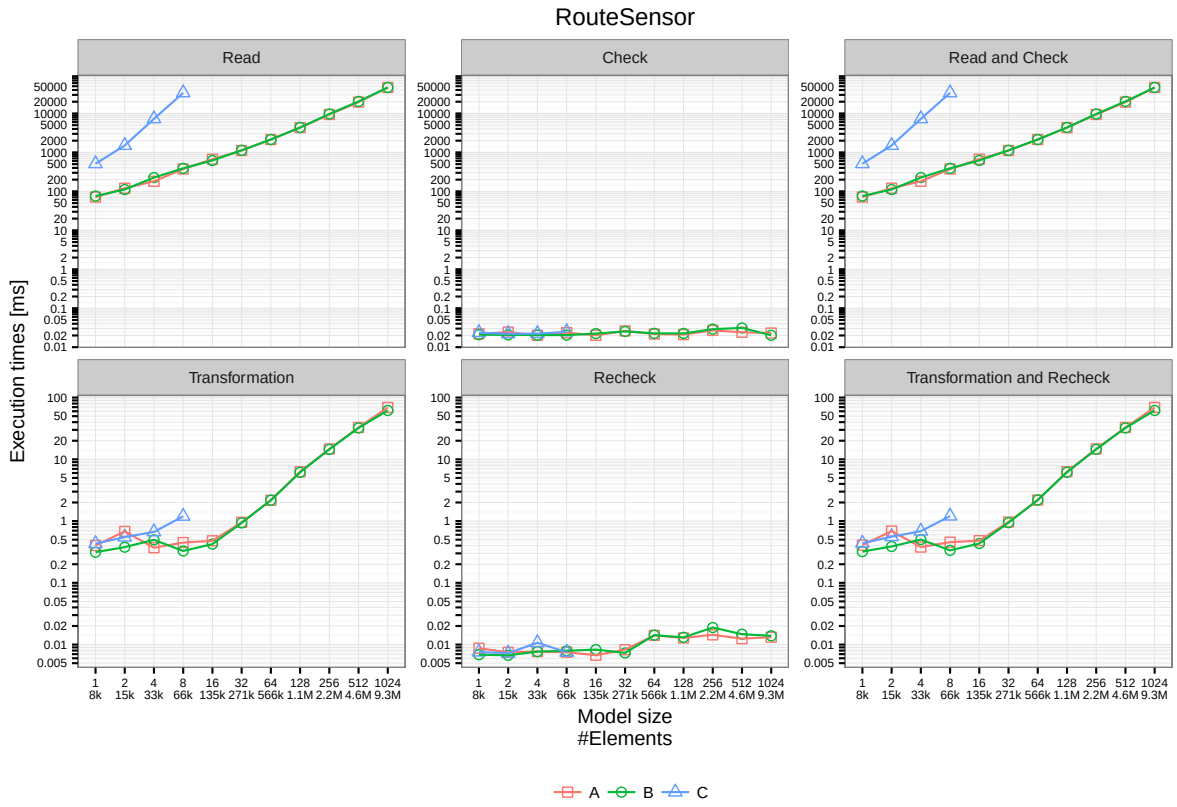

Figure 10: Execution times for the RouteSensor query variants (*A*–*C*).

Figure 10 and Figure 11 show the results of the benchmarks. The *x*-axis shows the model size (number of triples), while the *y*-axis shows the time required for each phase. Both axes use logarithmic scale.

RouteSensor. Variants *A* and *B* provide similar performance. However, variant *C*, which requires the computation of a Cartesian product, shows much worse scalability characteristics, only scaling for small models (up to 66k elements). Also, both the batch validation (Read and Check) and the incremental revalidation (Transformation and Recheck) are slower for variant *C* than for the other variants.

SemaphoreNeighbor. Variants *A*–*D* show similar performance, despite some differences in their query plans. The performance of variant *E* is significantly worse, while variant *F* offers the worst performance among the query plans, only scaling for small models (with 15k elements).

Analysis. The results show that different incremental graph query plans that avoid Cartesian products do not show significant differences in performance. A possible ex-

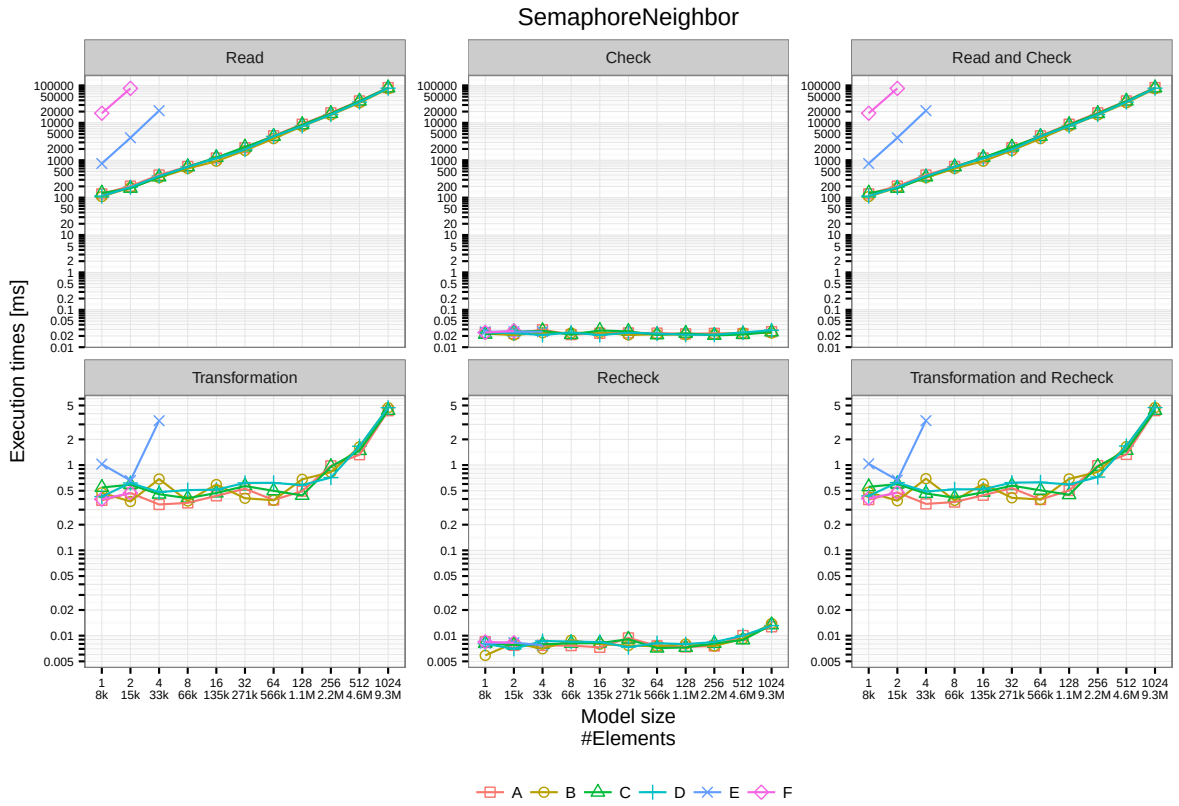

Figure 11: Execution times for the SemaphoreNeighbor query variants (A–F).

planation for this is that for query plans avoiding Cartesian products, indexing the graph dominates the execution time. After the graph is indexed, join and filtering operations can be calculated quickly (compared to indexing), even if their ordering is not optimal.

6.3 Utilization of the Results

The results suggest that similarly to relational query optimization engines, graph query optimization engines should focus on avoiding Cartesian products in the query plan. In the future, we plan to implement a graph query optimizer utilizing design space exploration (DSE) techniques, e.g. VIATRA-DSE [1] engine, which will feature both heuristic optimization techniques (e.g. pushing selection operators closer to leaf nodes) and cost-based query plans transformations.

7 Related Work

Graph query optimization techniques have been proposed for various algorithms and technological spaces. We discuss both non-incremental and incremental approaches.

7.1 Non-Incremental Approaches

Database technologies. Zhao et al. [43] proposed a sophisticated query optimization method for graph queries. The optimization uses neighborhood and path analysis for speeding up the queries. Incremental queries were not considered in the paper, but were listed as future work. Krause et al. [16] defined an SQL-based query language for graph pattern matching. For evaluating graph queries, their approach uses SAP HANA, a relational database, including its optimization engine.

Model-driven technologies. The Fujaba [22] graph transformation tool performs local search starting from the vertex selected by the system designer and extends the matching step-by-step by neighbouring vertices and edges. Fujaba fixes a single, breadth-first traversal strategy at compile-time, using simple heuristics, e.g. that navigation along an edge with an at most one multiplicity constraint precedes navigations along edges with arbitrary multiplicity. PROGRES [29] uses a sophisticated cost model for basic operations and generates the search plan at compile-time by a greedy algorithm.

The approach of G. Varró et al. [38] uses both metamodel- and instance model-level information to adaptively optimize graph queries based on statistical data collected from the current instance model. GrGen.NET [11] provides a dynamic, runtime optimization engine, which uses a mix of heuristical and cost-based techniques [11]. Recently, G. Varró et al. [39] proposed a dynamic programming-based algorithm using model statistics for optimizing search plans for pattern matching on EMF models.

7.2 Incremental Approaches

Rete algorithm for graph queries. The Rete algorithm was originally created by Charles Forgy for rule-based expert systems [9]. Bunke et al. [6] were the first to propose the Rete algorithm in the context of graph transformations.

Incremental algorithms. The TREAT algorithm aims at minimizing memory usage, while having the same algorithmic complexity as Rete. It stores only the input facts (input relations) and the conflict sets, and does not store partial pattern matches. Another improvement of Rete is the LEAPS algorithm [31], which aims to provide better space-time complexity. Rete itself has many improved versions (e.g. Rete II, Rete III, Rete-NT), however, unlike the original algorithm, these are not publicly available. Gator is a generalization the Rete and TREAT algorithms [5]. The authors of [12] optimized Gator networks using randomized state-space searching algorithms, which turned out to be superior to dynamic programming approaches in their use cases.

Doorenbos [7] proposed runtime optimization techniques, including left and right unlinking which aim to minimize to communication in the network. G. Varró et al. [37] presented an algorithm for constructing Rete networks with the goal of minimizing the size of the Rete network, using dynamic programming for identifying shared subpatterns.

Bergmann et al. adapted and improved the Rete algorithm for the Eclipse Modeling Framework (EMF) [2] in the EMF-INCQUERY project [3], now called VIATRA Query [36]. This approach provides VIATRA Query Language (VQL), a high-level declarative language for specifying queries. The query engine generates the query plan from the query specification using a greedy algorithm and utilizes subpattern reuse.

Rule engines. Drools [24], a business rule management system uses PHREAK, an enhanced version of the Rete algorithm, including a lazy loading optimization technique, which is beneficial for knowledge bases with a lot of rules/queries. Ishida [15] presented optimization techniques for Rete-based rule engines using a large number of rules.

Semantic technologies. Rete-based query evaluation is used for processing Linked Data as well. INSTANS [25] uses this algorithm to perform complex event processing on streaming RDF [41]. Diamond [21] also uses a Rete network to evaluate SPARQL queries on RDF data sets. Also, Peters et al. [23] proposed a Rete-based, parallel rule engine, which utilizes GPUs to parallelize transformations on RDF data.

Distributed graph queries. Some authors of this paper designed and implemented a distribute Rete-based incremental graph query engine, INCQUERY-D [31], which relies on the optimizer of EMF-INCQUERY and is able to scale for large models.

8 Conclusion and Future Work

In this paper, we presented optimization methods for incremental graph query plans, performed benchmarks and discussed the effect of optimization on the query execution time and scalability. The results show that query plans that unnecessary calculate Cartesian products slow down the evaluation by orders of magnitude. However, query plans without unnecessarily Cartesian products provide acceptable evaluation times and quick reevaluation. Reordering join and filtering operations in such plans only results in minor differences in execution times.

For future work, we plan to use multidimensional graph metrics [32] for optimizing query plans. Also, we plan to run performance experiments with query and transformation mixes, i.e. multiple queries and transformations executed at once.

Acknowledgements

This work was partially supported by the MTA-BME Lendület Research Group on Cyber-Physical Systems.

We would like to thank Gábor Bergmann and István Ráth for their suggestions. We are also thankful to József Marton for his comments on the draft of this paper, and to DigitalOcean, Inc. (`http://digitalocean.com/`) for generously providing cloud virtual machines for executing the benchmarks.

References

[1] H. Abdeen, D. Varró, H. A. Sahraoui, A. S. Nagy, C. Debreceni, Á. Hegedüs, and Á. Horváth. Multi-objective optimization in rule-based design space exploration. In *ASE*, pages 289–300, 2014. doi: 10.1145/2642937.2643005.

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

[3] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös. Incremental evaluation of model queries over EMF models. In *MODELS*, pages 76–90, 2010. doi: 10.1007/978-3-642-16145-2_6.

[4] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró. A Graph Query Language for EMF models. In *Fourth International Conference on Theory and Practice of Model Transformations*, volume 6707 of *LNCS*, pages 167–182. Springer, June 2011.

[5] T. Beyhl, D. Blouin, H. Giese, and L. Lambers. On the operationalization of graph queries with generalized discrimination networks. In *ICGT*, pages 170–186, 2016. doi: 10.1007/978-3-319-40530-8_11.

[6] H. Bunke, T. Glauser, and T. Tran. An efficient implementation of graph grammars based on the RETE matching algorithm. In *Graph-Grammars and Their Application to Computer Science, 4th International Workshop*, pages 174–189, 1990. doi: 10.1007/BFb0017389.

[7] R. B. Doorenbos. *Production matching for large learning systems*. PhD thesis, University of Southern California, 1995.

[8] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 3rd Edition*. Addison-Wesley-Longman, 2000. ISBN 978-0-8053-1755-8.

[9] C. Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1):17–37, 1982. doi: 10.1016/0004-3702(82)90020-0.

[10] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems – The complete book*. Pearson Education, 2nd edition, 2009. ISBN 978-0-13-187325-4.

[11] R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *ICGT*, pages 383–397, 2006. doi: 10.1007/11841883_27.

[12] E. N. Hanson, S. Bodagala, and U. Chadaga. Trigger condition testing and view maintenance using optimized discrimination networks. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):261–280, 2002. doi: 10.1109/69.991716.

[13] M. E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980. doi: 10.1109/TIT.1980.1056220.

[14] J. Hölsch and M. Grossniklaus. An algebra and equivalences to transform graph patterns in Neo4j. In *GraphQ workshop at EDBT/ICDT*, 2016.

[15] T. Ishida. An optimization algorithm for production systems. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):549–558, 1994. doi: 10.1109/69.298172.

[16] C. Krause, D. Johannsen, R. Deeb, K. Sattler, D. Knacker, and A. Niadzelka. An SQL-based query language and engine for graph pattern matching. In *ICGT*, pages 153–169, 2016. doi: 10.1007/978-3-319-40530-8_10.

[17] J. Maginecz and G. Szárnyas. Sharded joins for scalable incremental graph queries. In *Proceedings of the 23rd PhD Mini-Symposium*, February 2016. ISBN 978-963-313-220-3.

[18] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983. ISBN 0-914894-42-0. URL `http://web.cecs.pdx.edu/~maier/TheoryBook/TRD.html`.

[19] J. Makai, G. Szárnyas, Á. Horváth, I. Ráth, and D. Varró. Optimization of incremental queries in the cloud. In *CloudMDE workshop at MODELS*, 2015.

[20] D. P. Miranker and B. J. Lofaso. The organization and performance of a TREAT-based production system compiler. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):3–10, Mar. 1991. ISSN 1041-4347. doi: 10.1109/69.75882.

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

[22] U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In *ICSE*, pages 742–745. ACM, 2000. ISBN 1-58113-206-9. doi: 10.1145/337180.337620.

[23] M. Peters, C. Brink, S. Sachweh, and A. Zündorf. Scaling parallel rule-based reasoning. In *ESWC*, pages 270–285, 2014. doi: 10.1007/978-3-319-07443-6_19.

[24] Red Hat. Drools. `http://www.drools.org/`.

[25] M. Rinne. SPARQL update for complex event processing. In *ISWC'12*, volume 7650 of *LNCS*, pages 453–456. 2012. ISBN 978-3-642-35172-3. doi: 10.1007/978-3-642-35173-0_38.

[26] M. A. Rodriguez and P. Neubauer. The graph traversal pattern. In *Graph Data Management: Techniques and Applications*, pages 29–46. 2011. doi: 10.4018/978-1-61350-053-8.ch002.

[27] G. Sadowski and P. Rathle. Fraud detection: Discovering connections with graph databases. Technical report, Neo Technology, 2014. White paper.

[28] S. Sagiroglu and D. Sinanc. Big data: A review. In *CTS*, pages 42–47, 2013. doi: 10.1109/CTS.2013.6567202.

[29] A. Schürr, A. J. Winter, and A. Zündorf. Handbook of graph grammars and computing by graph transformation. pages 487–550. World Scientific Publishing Co., Inc., 1999. ISBN 981-02-4020-1. URL `http://dl.acm.org/citation.cfm?id=328523.328617`.

[30] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts, 5th Edition*. McGraw-Hill Book Company, 2005. ISBN 978-0-07-295886-7.

[31] G. Szárnyas, B. Izsó, I. Ráth, D. Harmath, G. Bergmann, and D. Varró. IncQuery-D: A distributed incremental model query framework in the cloud. In *MODELS*, pages 653–669, 2014. doi: 10.1007/978-3-319-11653-2_40.

[32] G. Szárnyas, Z. Kővári, Á. Salánki, and D. Varró. Towards the characterization of realistic models: Evaluation of multidisciplinary graph metrics. In *MODELS*, 2016.

[33] G. Szárnyas, B. Izsó, I. Ráth, and D. Varró. The Train Benchmark: Cross-technology performance evaluation of continuous model validation. *Software and Systems Modeling*, 2017. Accepted.

[34] A. Taylor and A. Jones. Cypher query language. `http://www.slideshare.net/graphdevroom/cypher-query-language`, 2012.

[35] Z. Ujhelyi, G. Szőke, Á. Horváth, N. I. Csiszár, L. Vidács, D. Varró, and R. Ferenc. Performance comparison of query-based techniques for anti-pattern detection. *Information & Software Technology*, 65:147–165, 2015. doi: 10.1016/j.infsof.2015.01.003.

[36] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software and Systems Modeling*, 15(3):609–629, 2016. doi: 10.1007/s10270-016-0530-4.

[37] G. Varró and F. Deckwerth. A Rete network construction algorithm for incremental pattern matching. In *ICMT*, pages 125–140, 2013. doi: 10.1007/978-3-642-38883-5_13.

[38] G. Varró, K. Friedl, and D. Varró. Adaptive graph pattern matching for model transformations using model-sensitive search plans. *Electronic Notes in Theoretical Computer Science*, 152:191–205, 2006. doi: 10.1016/j.entcs.2005.10.025.

[39] G. Varró, F. Deckwerth, M. Wieber, and A. Schürr. An algorithm for generating model-sensitive search plans for pattern matching on EMF models. *Software and Systems Modeling*, 14(2):597–621, 2015. doi: 10.1007/s10270-013-0372-2.

[40] J. Whittle, J. E. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014. doi: 10.1109/MS.2013.65.

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

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

[43] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3 (1):340–351, 2010.

A Rete Layouts

Figure 12–20) show possible Rete layouts for the RouteSensor and SemaphoreNeighbor queries.

Input nodes are marked with dashed lines, while *worker nodes* are marked with solid lines. Due to the sake of conciseness, *production nodes* were omitted in the figures. All Rete networks have a single production node as a parent of their depicted root node.

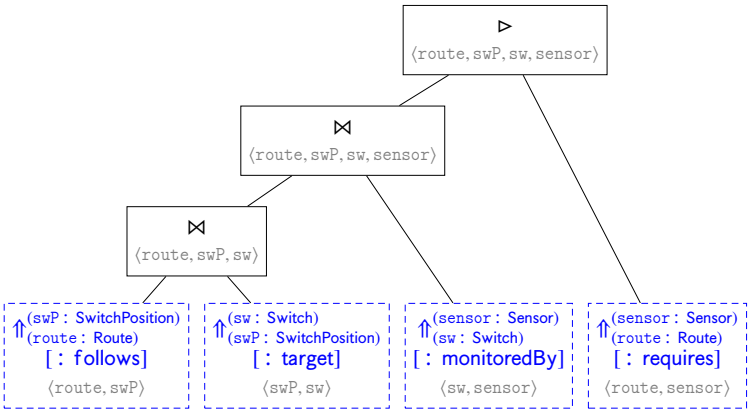

Figure 12: Rete layout *A* for pattern RouteSensor.

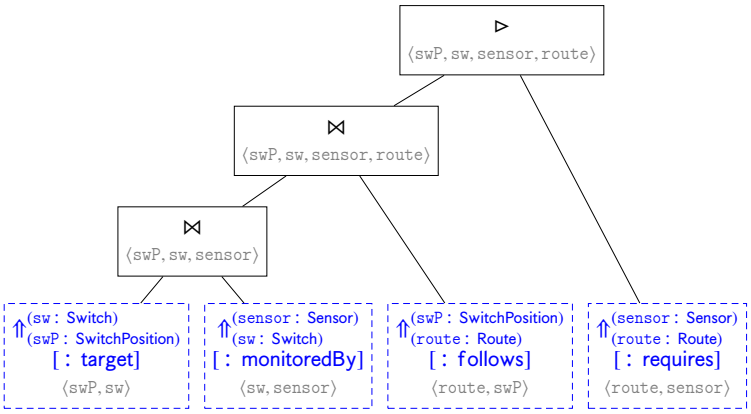

Figure 13: Rete layout *B* for pattern RouteSensor.

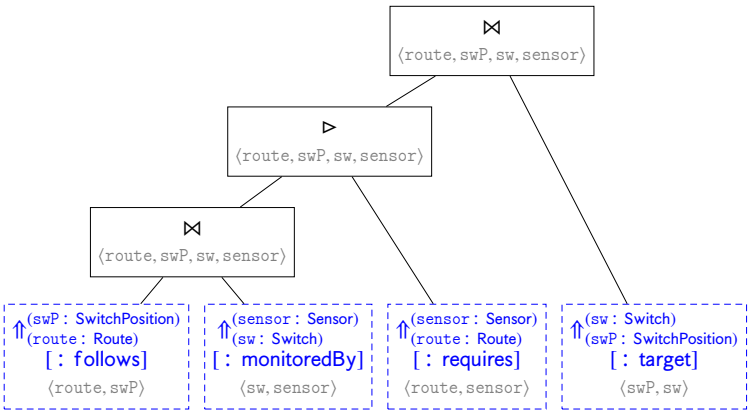

Figure 14: Rete layout C for pattern RouteSensor.

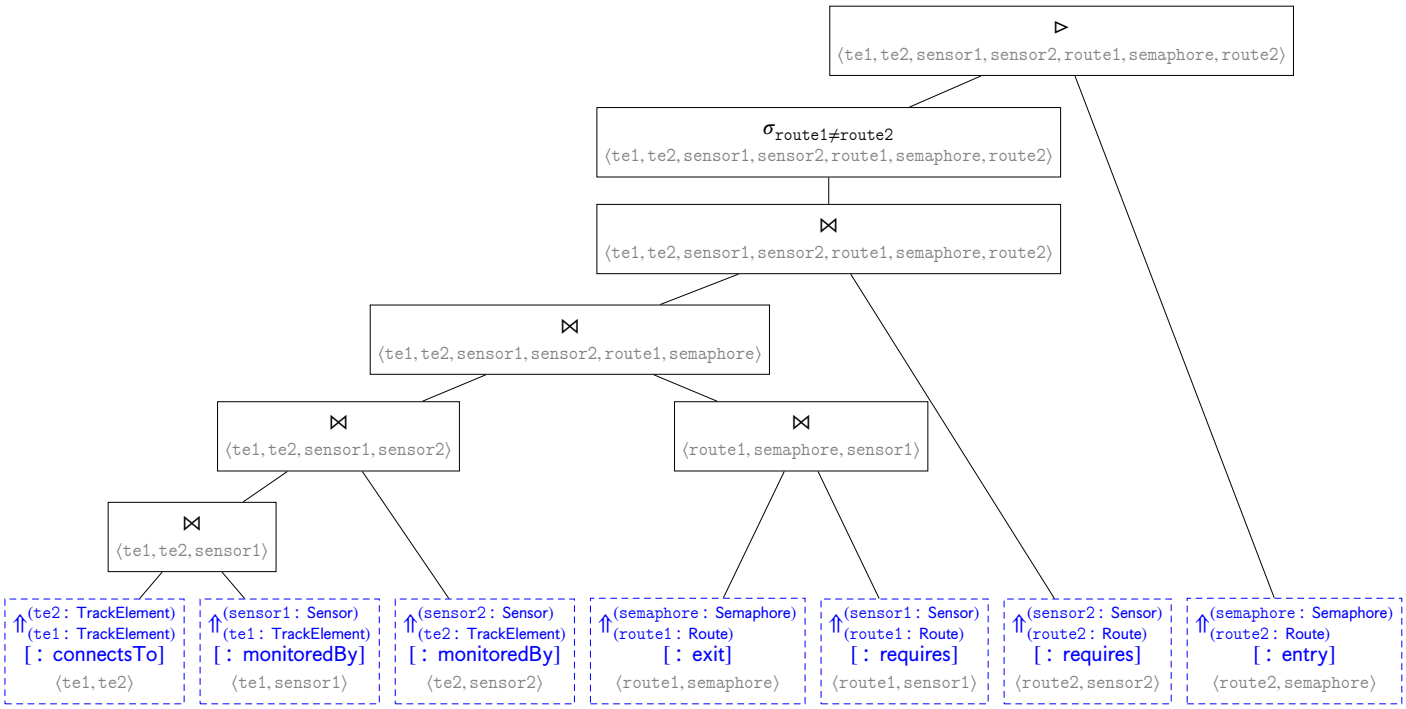

Figure 15: Rete layout A for pattern SemaphoreNeighbor.

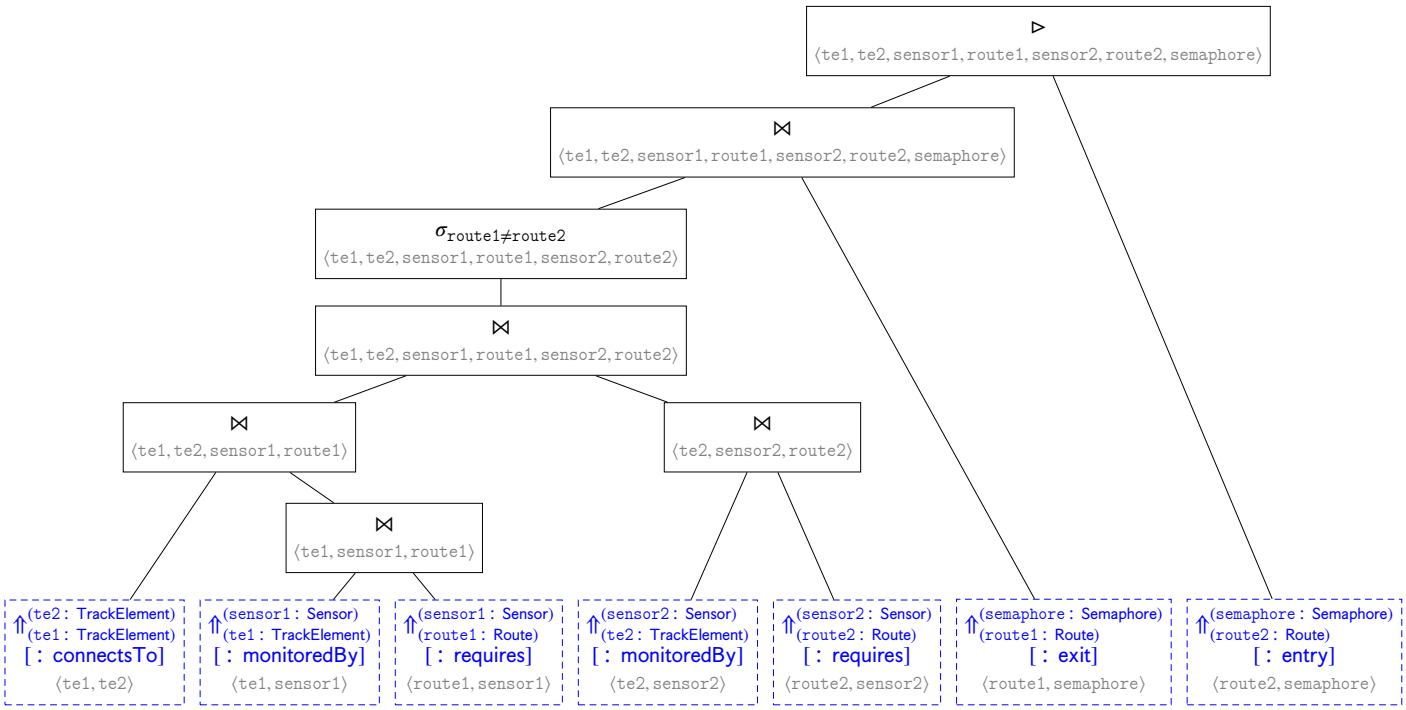

Figure 16: Rete layout *B* for pattern SemaphoreNeighbor.

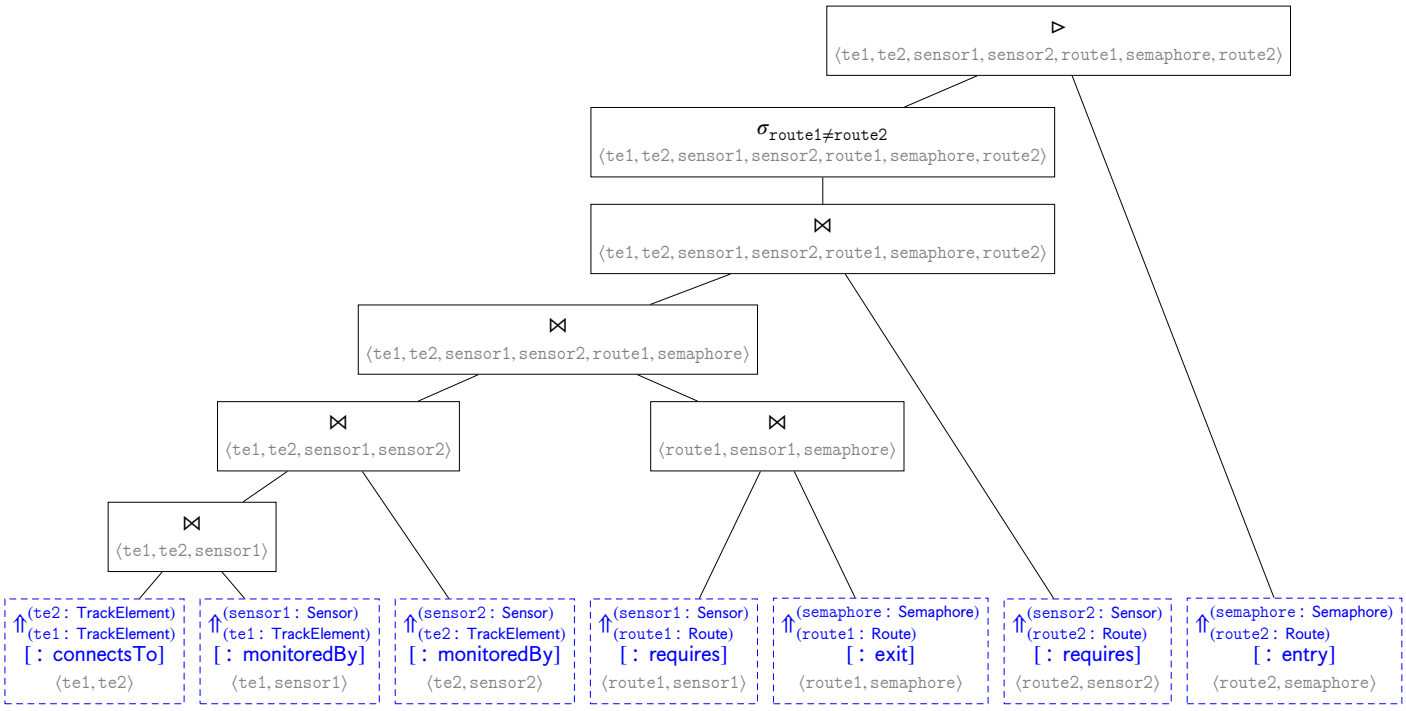

Figure 17: Rete layout *C* for pattern SemaphoreNeighbor.

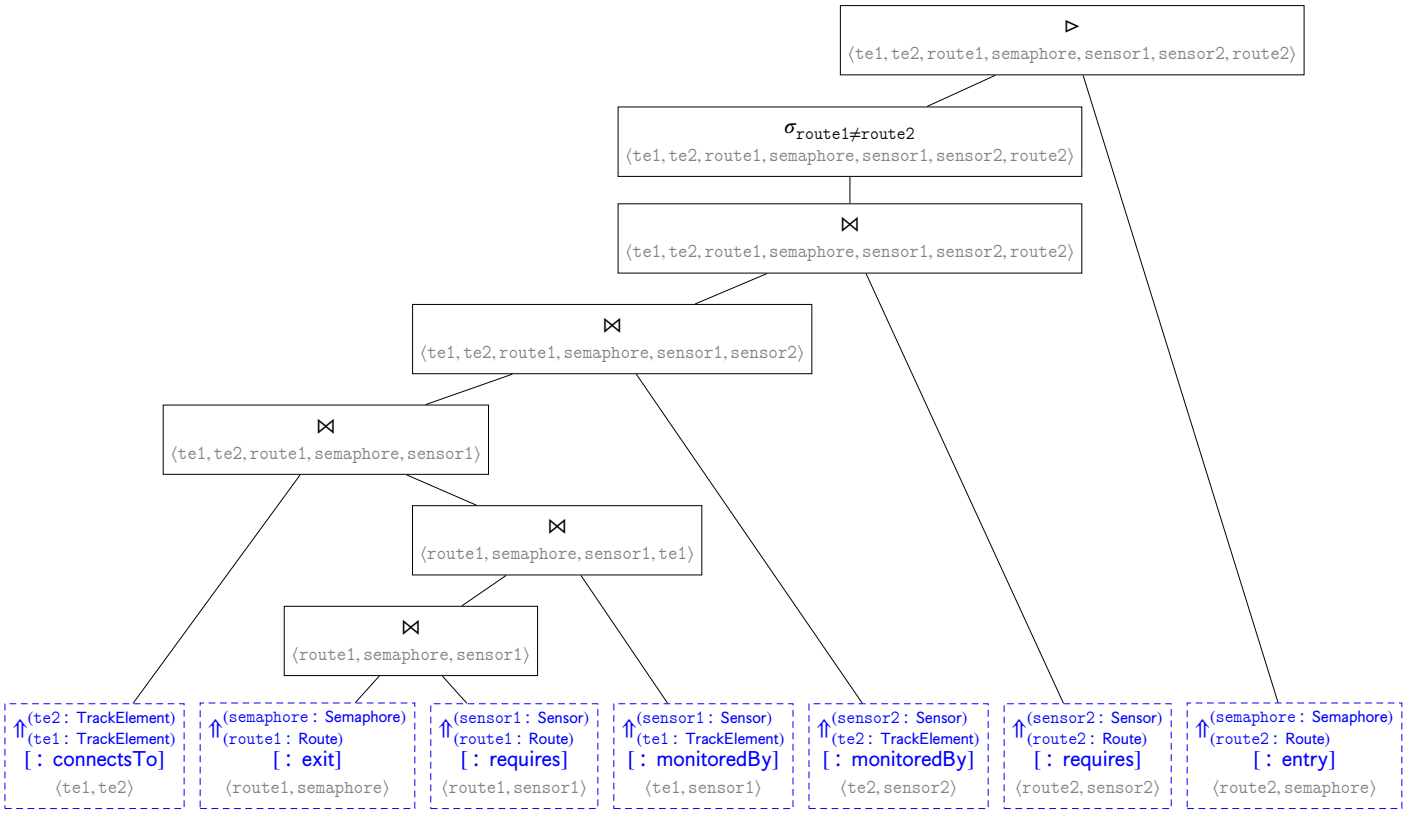

Figure 18: Rete layout D for pattern SemaphoreNeighbor.

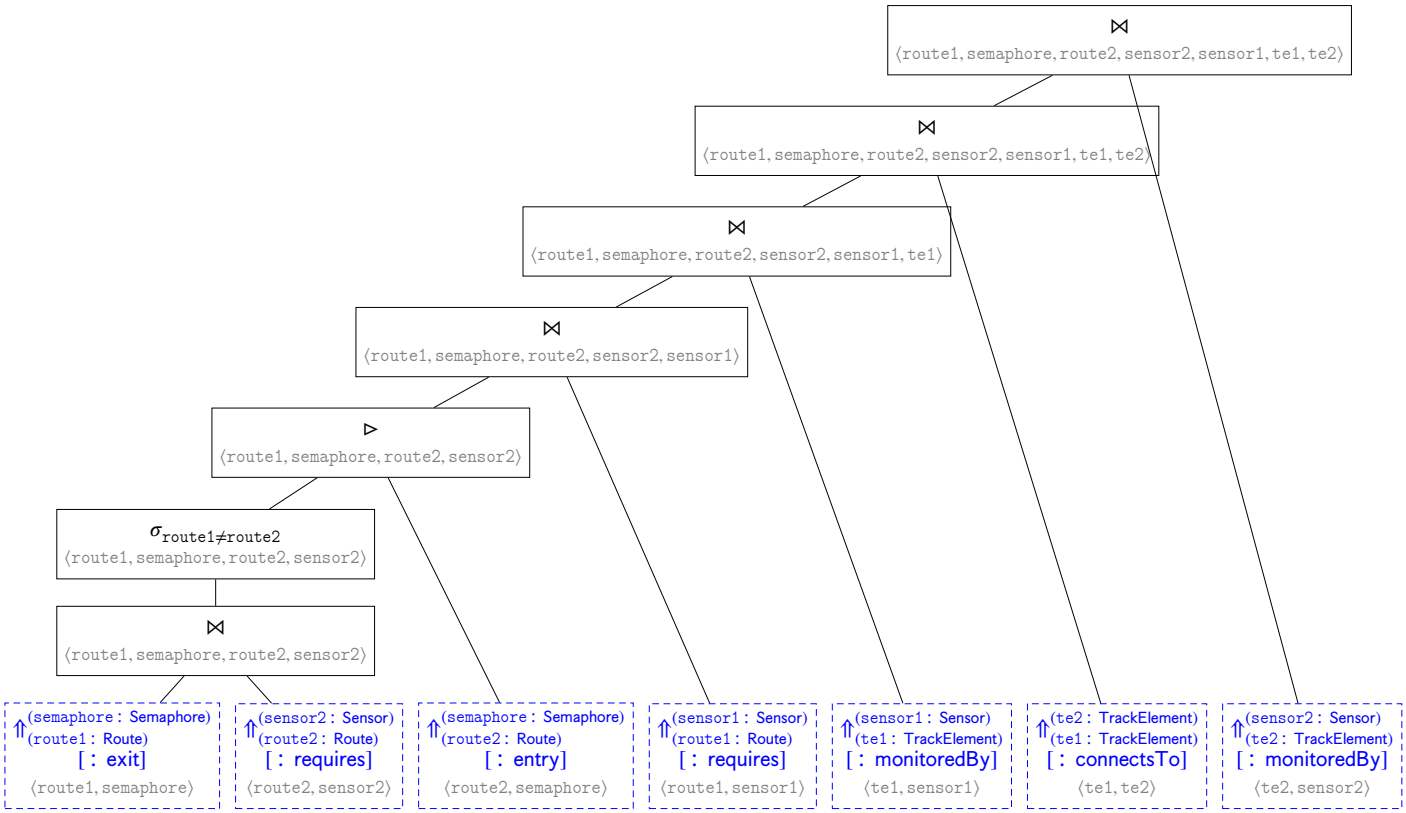

Figure 19: Rete layout E for pattern SemaphoreNeighbor.

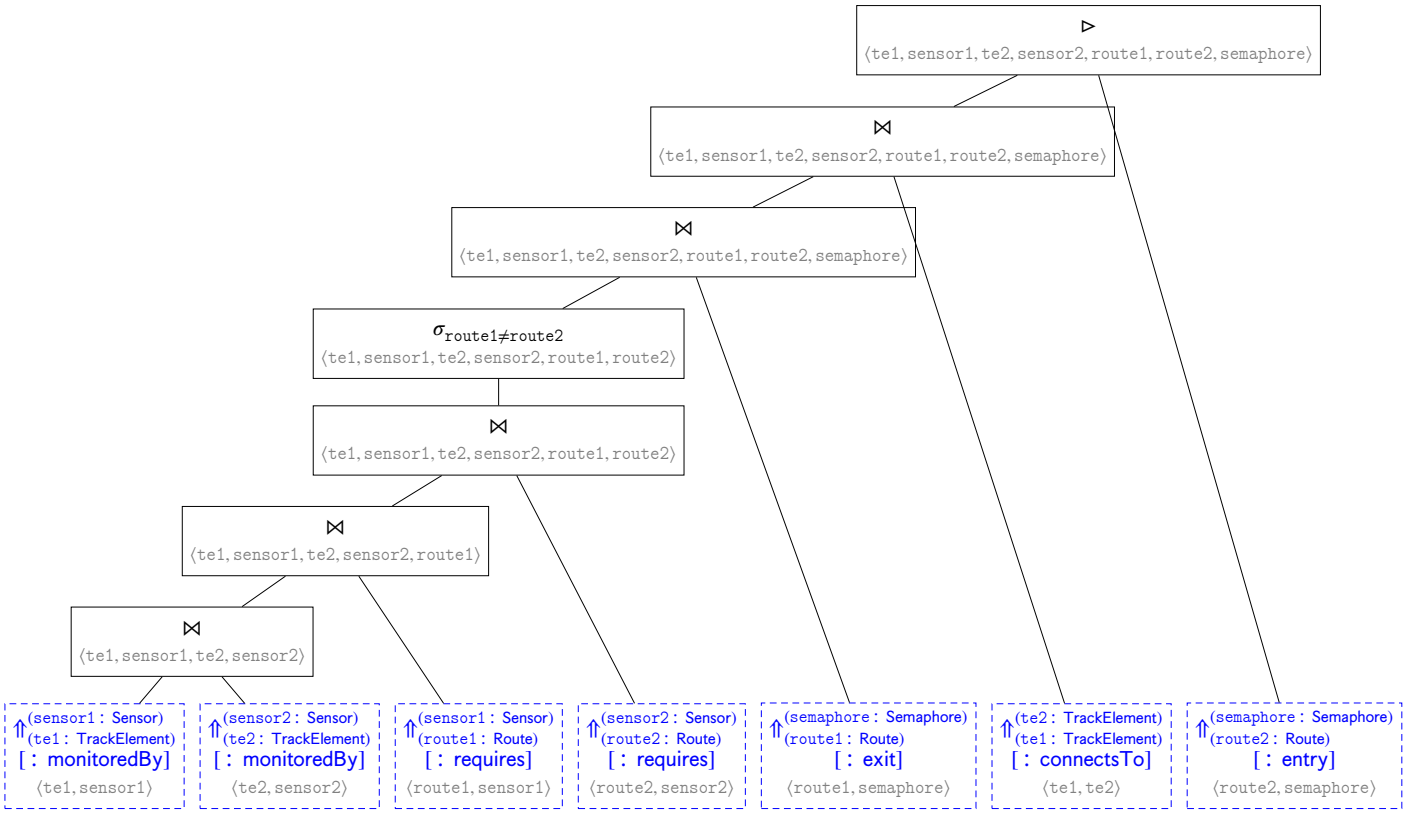

Figure 20: Rete layout F for pattern SemaphoreNeighbor.