Model-driven engineering of an openCypher engine: using graph queries to compile graph queries

József Marton[1], Gábor Szárnyas[2,3], and Márton Búr[2,3]

[1] Budapest University of Technology and Economics, Database Laboratory
`marton@db.bme.hu`
[2] Budapest University of Technology and Economics
Fault Tolerant Systems Research Group
[3] MTA-BME Lendület Research Group on Cyber-Physical Systems
`{szarnyas, bur}@mit.bme.hu`

The final publication is available at Springer via
`https://doi.org/10.1007/978-3-319-68015-6_6`.

Abstract. Graph database systems are increasingly adapted for storing and processing heterogeneous network-like datasets. Many challenging applications with near real-time requirements—such as financial fraud detection, on-the-fly model validation and root cause analysis—can be formalised as graph problems and tackled with graph databases efficiently. However, as no standard graph query language has yet emerged, users are subjected to the possibility of vendor lock-in.
The openCypher group aims to define an open specification for a declarative graph query language. However, creating an openCypher-compatible query engine requires significant research and engineering efforts. Meanwhile, model-driven language workbenches support the creation of domain-specific languages by providing high-level tools to create parsers, editors and compilers. In this paper, we present an approach to build a compiler and optimizer for openCypher using model-driven technologies, which allows developers to define declarative optimization rules.

1 Introduction

Context. Graphs provide an intuitive formalism for modelling real-world scenarios, as the human mind tends to interpret the world in terms of objects (*vertices*) and their respective relationships to one another (*edges*) [30].

The *property graph* data model [33] extends graphs by adding labels/types and properties for vertices and edges. This gives a rich set of features for users to model their specific domain in a natural way. Graph databases are able to store property graphs and query their contents by matching complex graph patterns, which would otherwise be cumbersome to define and/or inefficient to evaluate on traditional relational databases [39].

Neo4j, a popular NoSQL property graph database, offers the Cypher query language to specify graph queries. Cypher is a high-level declarative query language, detached from the query execution plan, which allows the query engine to use sophisticated optimisation techniques. The openCypher project [25] aims to deliver an open specification of Cypher.

Problem and objectives. Even though the openCypher specification was released more than 1.5 years ago, there are very few open implementations available and even those offer limited support for the more advanced language constructs. Besides the novelty of the openCypher specification, the primary reason for the lack of open implementations is the complexity of the language. Even with the artifacts provided by the openCypher project—including the specification, the language grammar and a set of test cases—implementing a compiler is a non-trivial task and requires significant engineering efforts. Our goal is to deliver a reusable compiler that can be extended with transformation rules for query optimisation.

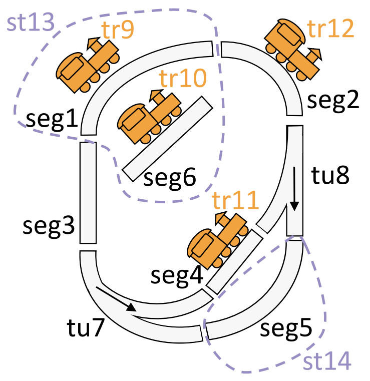
Contributions. In this paper, we use graph queries defined on a cyber-physical system to demonstrate the key challenges in compiling openCypher queries. We present an approach for implementing an openCypher query compiler including a model-based parser generator and a set of model transformation rules built on a modern language workbench based on Eclipse technologies. We released the compiler as part of the open-source ingraph project, where it is used as part of an incremental graph query engine, released under the commercially-friendly Eclipse Public License.[4]

Structure of the paper. We first introduce the running example in Sec. 2 and the concepts of graph queries and model transformations in Sec. 3. We give an overview of the compiler in Sec. 4 and use example queries to elaborate the details of query compilation in Sec. 5. We discuss related research in Sec. 6 and conclude the paper in Sec. 7.

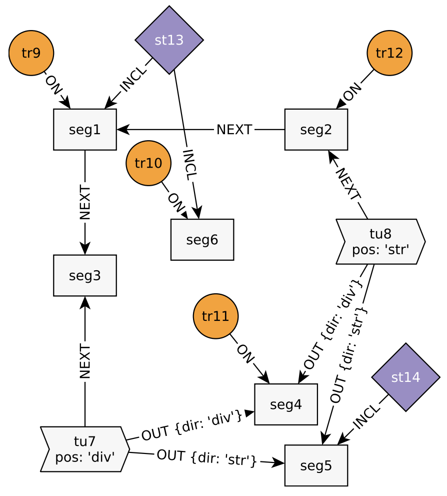
2 Running Example

To demonstrate our approach, we use a cyber-physical system demonstrator, MoDeS3 [7], which stands for Model-Based Demonstrator for Smart and Safe Systems. It is an educational platform of a model railway system that prevents trains from collision and derailment using runtime verification techniques based on safety monitors. The railway track is instrumented with several sensors, such as *cameras* and *shunt detectors* capable of sensing trains on a particular segment of a track, connected to computing units. In addition to collecting data, these computing units also control the trains to guarantee safe operation. In this paper, we will only introduce a small self-contained fragment of the demonstrator in order to keep the example compact.

[4] Available at `http://docs.inf.mit.bme.hu/ingraph/`.



(a) MoDeS3 example graphical syntax.



(b) MoDeS3 example graph.

Fig. 1: The running example.

Fig. 1(a) depicts a snapshot of the simplified system in operation, where trains are located at different parts of the railway. The railway network itself consists of two types of railway elements: *segments* and *turnouts*. Segments are selected tracks of the railway network with one entry and exit points individually, they are approximately of same lengths, and they have no intermediate branches between the entry and exit points. As opposed to segments, turnouts allow trains to change tracks. A turnout can either be in *divergent* or *straight* state. A *station* can represent a railway station with an arbitrary purpose, and they can include any number of railway elements.

We introduce the following example *monitoring objectives* that are evaluated continuously by graph queries:

– *Close proximity* identifies trains on consecutive segments with only a limited distance between each other (train `tr9` on `seg1` and `tr12` on `seg2` in the example).
– *Station with free track* monitoring objective finds stations that have at least one free track available (station `st14` in the example).
– *Busy station* identifies stations with at least two trains residing on its corresponding tracks (station `st13` in the example).

3 Preliminaries

In this section, we present the theoretical and practical foundations for compiling openCypher queries. This includes the notion of property graphs, a brief description of the openCypher language and the relational algebraic foundations for formalising graph queries. We also discuss model-driven engineering (MDE) along with the MDE tools used in our work.

3.1 Property Graphs and the openCypher Query Language

The *property graph* data model [32] extends typed graphs with properties on the vertices and edges. This data model is used in NoSQL graph database systems such as Neo4j [24], OrientDB [27], SparkSee [36], and Titan [40]. Graph databases provide no or weak metamodeling capabilities. Hence, models can either be stored in a weakly typed manner or the metamodel must be included in the graph (on the same metalevel as the instance model). The property graph of the running example is shown in Fig. 1(b).

Cypher is a high-level declarative graph query language used in the Neo4j graph database [29]. It allows users to specify graph patterns with a syntax resembling an actual graph, which makes the queries easy to comprehend. The goal of the openCypher project [25] is to provide a standardised specification of the Cypher language.

List. 3.1 shows a query that returns all `tr`, `seg` pairs, where a particular train `tr` is `ON` a particular segment `seg`.

```
MATCH (tr:Train)-[:ON]->(seg:Segment)
RETURN tr, seg
```

List. 3.1: Example openCypher query.

3.2 Relational Graph Algebra

We gave a formal specification for the core subset of the openCypher language in [23] using relational graph algebra, which extends relational algebra with graph-specific operators. Here, we give a brief summary of the operators in relational graph algebra, which operates on multisets (bags) [15] of tuples, that form *graph relations*. We refer to named elements of a tuple as *attributes*.

Notation. Graph relations, schemas and attributes are typeset in *italic* (r, R, A_1), variable names set in monospace (`x1`), while labels, types and constants are set in sans-serif (min, l_1, t_k). The NULL value is represented as ε.

Nullary operators. The *get-vertices* [18] nullary operator $\bigcirc_{(\mathtt{v}:\mathsf{l}_1 \wedge \ldots \wedge \mathsf{l}_\mathsf{n})}$ returns a graph relation of a single attribute v that contains vertices that have *all* of labels $\mathsf{l}_1, \ldots, \mathsf{l}_\mathsf{n}$.

Additionally to our previous work, we introduce *Dual*, which is a relation with no columns and a single (empty) tuple, i.e. $\mathit{Dual} = \{\langle\rangle\}$.[5] The *Dual* relation is the identity element of the Cartesian product and the natural join operators. We also introduce *Singular*, which denotes the empty relation $\{\}$ and is the zero element of the Cartesian product and the natural join operators.

[5] The *Dual* relation is inspired by the `DUAL` table in the Oracle database [6].

Unary operators. The *projection* operator π keeps the specified set of attributes of the relation: $\pi_{A_1,\ldots,A_n}(r)$. The projection operator can also rename attributes, e.g. $\pi_{\mathtt{x1}\rightarrow\mathtt{x2}}(r)$ renames `x1` to `x2`. Note that tuples are not deduplicated, i.e. the result has the same number of tuples as the input relation r.

As relational graph algebra operates on multisets, there is a bespoke operator for removing duplicate tuples. The *duplicate-elimination* operator δ takes a multiset of tuples on its input, performs deduplication and returns a set of tuples.

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

The *expand-out* unary operator $\uparrow_{(\mathtt{v})}^{(\mathtt{w}:\mathsf{l_1}\wedge\ldots\wedge\mathsf{l_n})}[\mathsf{e}:\mathsf{t_1}\vee\ldots\vee\mathsf{t_k}](r)$ adds new attributes e and w to each tuple iff there is an outgoing edge e from v to w, where e has *any* of types $\mathsf{t_1},\ldots,\mathsf{t_k}$, while w has *all* labels $\mathsf{l_1},\ldots,\mathsf{l_n}$. Similarly, the *expand-in* operator $\downarrow$ uses incoming edges, while the *expand-both* operator $\updownarrow$ uses both incoming and outgoing edges. An extended version of this operator, $\uparrow_{(\mathtt{v})}^{(\mathtt{w})}[\mathsf{e}*_{\mathsf{min}}^{\mathsf{max}}]$ may use any number of hops between min and max.

Binary operators. The result of the *natural join* operator $\bowtie$ is determined by creating the Cartesian product of the relations, then filtering for those tuples which are equal on the attributes that share a common name. The combined tuples are projected: for input relations r and s (with schemas R and S, respectively), we only keep the attributes in r and drop the ones in s. Hence,

$$r \bowtie s = \pi_{R\cup S}\ \sigma_{(r.A_1=s.A_1\ \wedge\ \ldots\ \wedge\ r.A_n=s.A_n)}(r \times s),$$

where $\{A_1,\ldots,A_n\} = R\cap S$ is the set of attributes that occur both in R and S.

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

$$r \rhd s = r \setminus \pi_R(r \bowtie s),$$

where π_R denotes a projection operation, which only keeps the attributes of the schema over relation r.

The *left outer join* $⟕$ pads tuples from the left relation that did not match any from the right relation with ε values and adds them to the result of the natural join [35].

Tab. 1 shows a concise set of rules for mapping openCypher expressions to relational graph algebra [23].

3.3 Model-Driven Engineering

Model-driven engineering (MDE) is a development paradigm, used in many areas of software and system engineering, such as designing safety-critical systems. MDE focuses on creating and analyzing models at different levels of abstraction during the engineering process. *Model transformations* are used to process models, e.g. to convert models between different modeling languages and to generate code.

Language construct	**Relational algebra expression**
Vertices and patterns. ⦇p⦈ denotes a pattern that contains a vertex «v».	
`(«v»:«l1»:⋯:«ln»)`	$\bigcirc_{(\mathtt{v}:\mathsf{l1}\wedge\cdots\wedge\mathsf{ln})}$
⦇p⦈`<-[«e»:«t1»|⋯|«tk»]->(«w»)`	$\updownarrow^{(\mathtt{w})}_{(\mathtt{v})}\,[\mathsf{e}:\mathsf{t1}\vee\cdots\vee\mathsf{tk}]\,(p)$, where e is an edge
⦇p⦈`-[«e»*«min»..«max»]->(«w»)`	$\uparrow^{(\mathtt{w})}_{(\mathtt{v})}\,[\mathsf{e}*^{\mathsf{max}}_{\mathsf{min}}]\,(p)$, where e is a list of edges
Combining and filtering pattern matches	
`MATCH` ⦇p1⦈, ⦇p2⦈, ⋯	$\not\equiv_{\text{edges of p1, p2, …}}(p1 \bowtie p2 \bowtie \cdots)$
`MATCH` ⦇p1⦈ `MATCH` ⦇p2⦈	$\not\equiv_{\text{edges of p1}}(p1) \bowtie \not\equiv_{\text{edges of p2}}(p2)$
`OPTIONAL MATCH` ⦇p⦈ `WHERE` ⦇condition⦈	$\mathit{Dual} \mathbin{⟕}_{\text{condition}} \not\equiv_{\text{edges of p}}(p)$
⟦r⟧ `OPTIONAL MATCH` ⦇p⦈	$\not\equiv_{\text{edges of r}}(r) \mathbin{⟕} \not\equiv_{\text{edges of p}}(p)$
⟦r⟧ `WHERE «condition»`	$\sigma_{\text{condition}}(r)$
⟦r⟧ `WHERE` ⦇p⦈	$r \ltimes p$
Result and subresult operations. Rules for `RETURN` also apply to `WITH`.	
⟦r⟧ `RETURN «x1» AS «y1», ⋯`	$\pi_{\mathtt{x1}\rightarrow\mathtt{y1},\cdots}(r)$
⟦r⟧ `RETURN «x1», «aggr»(«x2»)`	$\gamma^{\mathtt{x1}}_{\mathtt{x1},\mathsf{aggr}(\mathtt{x2})}(r)$
⟦r⟧ `WITH «x1»` ⟦s⟧ `RETURN «x2»`	$\pi_{\mathtt{x2}}\Big(\big(\pi_{\mathtt{x1}}(r)\big) \bowtie s\Big)$

Table 1: Mapping from openCypher constructs to relational algebra [23]. Variables, labels, types and literals are typeset as «v». The notation ⦇p⦈ represents patterns resulting in a relation p, while ⟦r⟧ denotes previous query fragment resulting in a relation r. To avoid confusion with the "`..`" language construct (used for ranges), we use ⋯ to denote omitted query fragments.

Domain-specific languages. While there are some extensible formalisms intended as a general-purpose way of representing models (such as UML), industrial practice often prefers domain-specific languages (DSLs) for describing modeling languages instead. These can be designed and modified to the needs of application domains and actual design processes. On the other hand, developing such a DSL (and providing tool support) is an expensive task.

The Eclipse Modeling Framework (EMF) is a domain-specific modeling technology, built on the Eclipse platform. A DSL development process with EMF starts with the definition of a metamodel, from which several components of the modeling tool can be automatically derived. The metamodel is defined in Ecore, the metamodeling language of EMF [37].

Language workbenches. Model-driven language workbenches [13] support the creation of domain-specific languages by providing high-level tools to create parsers, editors and compilers. Xtext [14] is an EMF-based framework for development of programming languages and DSLs. Xtend is a general-purpose programming language (implemented with an Xtext-based parser), which is transpiled to Java source code. Xcore [12] is an extended textual syntax for Ecore and provides an Xtext-based language for defining EMF metamodels.

Model transformations VIATRA [43] is an open-source Eclipse project written in Java and Xtend [11]. VIATRA builds on the Eclipse Modeling Framework and provides the following main features:

– The VIATRA Query Language, a declarative language for writing queries over models, which are evaluated once or incrementally upon each model change.
– An internal domain-specific language over the Xtend language to specify both batch and event-driven, reactive transformations.
– A rule-based design space exploration framework [17] to explore design candidates with transformation rules where the design candidates must satisfy multiple criteria.

4 Overview of the Approach

The high-level workflow of our openCypher query engine is shown in Fig. 2. A domain expert first formulates the *query* using the openCypher language, which serves as the input for our engine. The query is then parsed and transformed into the *query syntax graph* using the openCypher grammar (created by the Slizaa project[6]). It is then compiled to our relational graph algebra model. This produces a canonical relational graph algebra representation to keep compiler code simple. The relational graph algebra representation is modified by the relational algebra optimizer. The resulting relational algebra model is then passed on to the query execution engine.

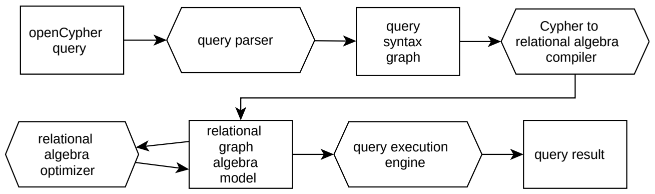

openCypher query
query parser
query syntax graph
Cypher to relational algebra compiler
relational algebra optimizer
relational graph algebra model
query execution engine
query result

Fig. 2: Workflow of the query engine: compiler and execution engine.

Relational graph algebra metamodel. The metamodel of the relational graph algebra operators introduced in Sec. 3.2 is shown in Fig. 3. An openCypher query is represented by a *rooted tree* having nullary operators as its leaves and unary or binary operators as its non-leaf nodes.

[6] `https://github.com/slizaa/slizaa-opencypher-xtext`, released under EPL v1.0.

Nullary operators. The GetVertices and GetEdges operators retrieve vertices and edges of the graph, respectively. SingularObjectSource and DualObjectSource emit the *Singular* and the *Dual* relation, respectively.

Unary operators. Projection and Selection work as given in Sec. 3.2. Exact semantics of the other unary operators are given in [23]. DuplicateElimination, Grouping, Sort and Top operators work like their corresponding `SQL` clauses.[7] Expand is a graph-specific operation to traverse one or a sequence of edges from a source to a given target vertex, while AllDifferent is specific to openCypher's edge uniqueness semantics. The Unwind operator is the inverse of the list-constructing `collect()` aggregation function.

Binary operators. The Union operator creates the set or multiset union of its inputs. Join, LeftOuterJoin and AntiJoin operators, based on the joinVariable list declared in AbtractJoin creates the natural join, antijoin and left outer join operations on their inputs, respectively, as given in Sec. 3.2.

Relational algebra optimizer. The relational algebra optimizer has two main tasks. It removes idempotent operations from the relational graph algebra model and identifies combinations of operations that could be expressed using advanced operations. The relational graph algebra model is also a graph, so both of these tasks are graph manipulation tasks which we have implemented using graph pattern matches using the Viatra model transformation framework (Sec. 3.3).

5 Elaboration

We have shown the overview of our approach in Sec. 4. In this section we present our approach in detail, driven by examples of the MoDeS3 system (Sec. 2). We focus on the *relational algebra optimizer*, and introduce the compiler to the extent needed to put the optimizations in context.

5.1 Compilation of a Multipart Query

In openCypher, queries are composed as a sequence of query parts. Details are given in [23], but essentially a query part contains clauses up to the next `WITH` or `RETURN` clause and defines a result set of the attributes listed, which is then fed into the next query part as its input. For example, the query in List. 5.1 is composed of two query parts: first query part spans lines 3–5 and feeds its result set of the schema $\langle s, countTrains \rangle$ into the second query part listed in line 6.

Variable chaining refers to the fact that attributes of the resulting schema are available in the subsequent query part, i.e. `s` and `countTrains` are available.

[7] In the order of appearance: `DISTINCT`, `GROUP BY`, `ORDER BY` and `SKIP ... LIMIT ...`

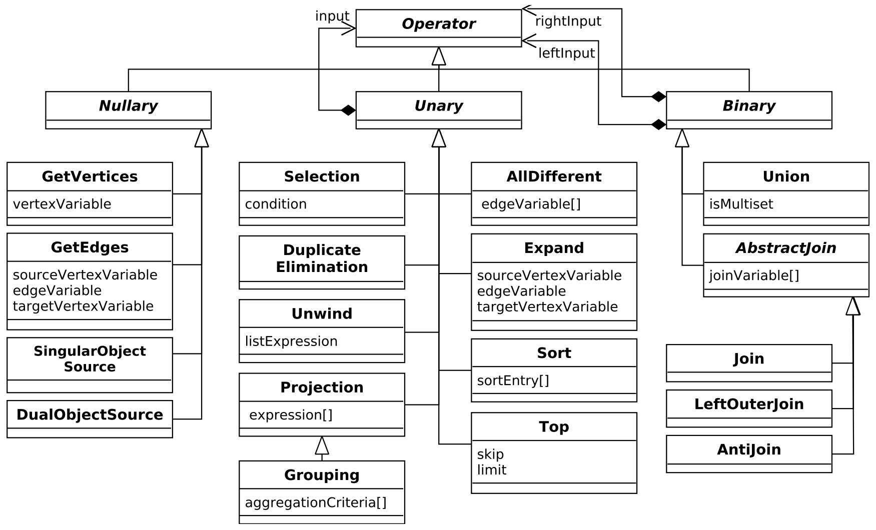


Fig. 3: Operator metamodel of the relational graph algebra.

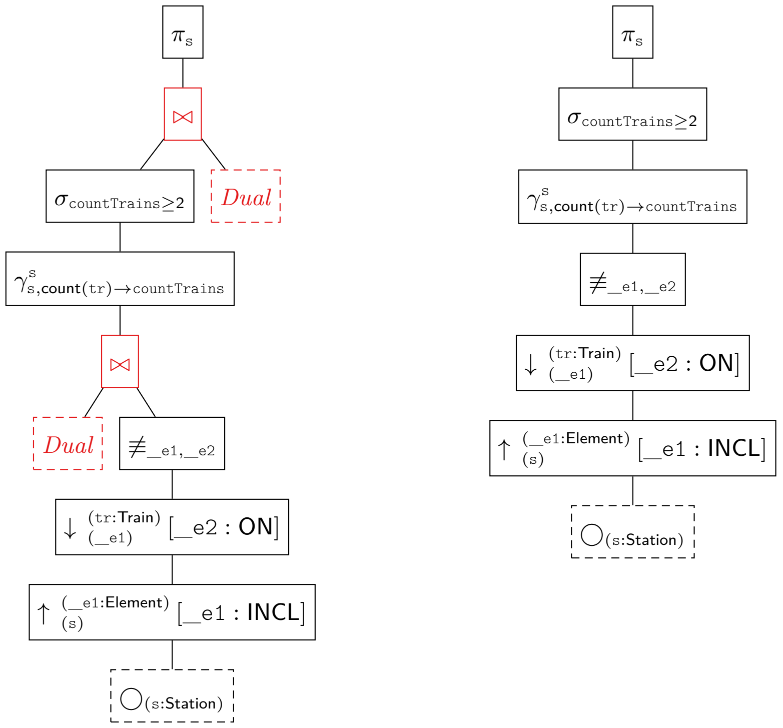
```
// identifies stations with at least two trains residing on its
// corresponding tracks
MATCH (s:Station)-[:INCL]->(:Element)<-[:ON]-(tr:Train)
WITH s, count(tr) AS countTrains
WHERE countTrains >= 2
RETURN s
```

List. 5.1: Busy station.

Compilation of each query part starts from the *Dual* relation. Each pattern given in a `MATCH` clause is then compiled and joined to the previous patterns: for `MATCH` clauses we use the natural join operator and for `OPTIONAL MATCH`, we use left outer join. Possible projection, grouping and duplicate-elimination operators are appended above as required by the `WITH` or `RETURN` clauses.

Query parts are compiled one by one and combined together using the natural join operator as follows. The natural join is injected into the compiled form of the current query part just below the possible projection, grouping and duplicate-elimination operators populating its right input with the descendants. Its left input is the compiled form of the query parts processed so far.

Each query part that begins with a non-optional `MATCH` clause, like the first query part in List. 5.1 is joined with *Dual*. As the second query part has no patterns, its inputs are the first query part's result set and the *Dual* relation. The raw compiled form of this query is shown in Fig. 4(a), which contains two joins having *Dual*, its identity operand as one of its operands. Thus these natural join operations along with *Dual* should be removed, which we implemented using a VIATRA graph transformation rule (see Sec. 5.4). Applying this transformation, we get the simplified form shown in Fig. 4(b).



π_s
$\bowtie$
$\sigma_{\text{countTrains}\geq 2}$
Dual
$\gamma^{s}_{s,\text{count(tr)}\rightarrow\text{countTrains}}$
$\bowtie$
Dual
$\not\equiv_{\_e1,\_e2}$
$\downarrow^{(tr:Train)}_{(\_e1)}[\_e2 : ON]$
$\uparrow^{(\_e1:Element)}_{(s)}[\_e1 : INCL]$
$\bigcirc_{(s:Station)}$
π_s
$\sigma_{\text{countTrains}\geq 2}$
$\gamma^{s}_{s,\text{count(tr)}\rightarrow\text{countTrains}}$
$\not\equiv_{\_e1,\_e2}$
$\downarrow^{(tr:Train)}_{(\_e1)}[\_e2 : ON]$
$\uparrow^{(\_e1:Element)}_{(s)}[\_e1 : INCL]$
$\bigcirc_{(s:Station)}$

(a) Raw query plan. (b) Simplified query plan.

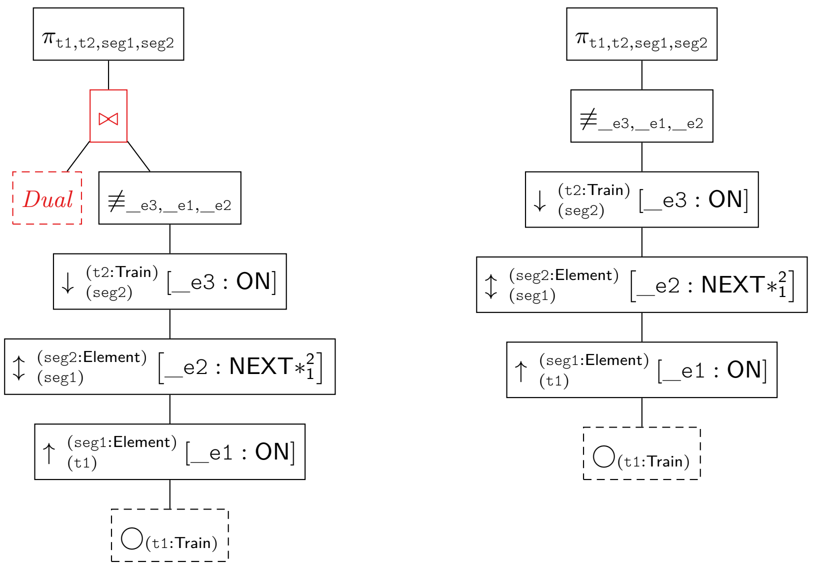
Fig. 4: Query plans for *Busy station.*

5.2 Compilation of Variable Length Path Patterns

```
// identify trains on consecutive segments with only a limited distance
// between each other
MATCH
  (t1:Train)-[:ON]->(seg1:Element)-[:NEXT*1..2]-
  (seg2:Element)<-[:ON]-(t2:Train)
RETURN t1, t2, seg1, seg2
```

List. 5.2: Close proximity.

The query in List. 5.2 features a variable length path pattern stating that two segments, seg1 and seg2 are connected through one to two edges of type NEXT. A variable length path pattern is compiled to an expand-both operator given in Sec. 3.2. The raw compiled form of this query is shown in Fig. 5(a), which is simplified to Fig. 5(b) using the transformation rule described in Sec. 5.1 to remove a join having *Dual* on one of its inputs.


$\pi_{\texttt{t1,t2,seg1,seg2}}$
$\bowtie$
Dual
$\not\equiv_{\_\_e3,\_\_e1,\_\_e2}$
$\downarrow$ (t2:Train) (seg2) [__e3 : ON]
$\updownarrow$ (seg2:Element) (seg1) [__e2 : NEXT$*_1^2$]
$\uparrow$ (seg1:Element) (t1) [__e1 : ON]
$\bigcirc$(t1:Train)
$\pi_{\texttt{t1,t2,seg1,seg2}}$
$\not\equiv_{\_\_e3,\_\_e1,\_\_e2}$
$\downarrow$ (t2:Train) (seg2) [__e3 : ON]
$\updownarrow$ (seg2:Element) (seg1) [__e2 : NEXT$*_1^2$]
$\uparrow$ (seg1:Element) (t1) [__e1 : ON]
$\bigcirc$(t1:Train)

(a) Raw query plan. (b) Simplified query plan.

Fig. 5: Query plans for *Close proximity*.

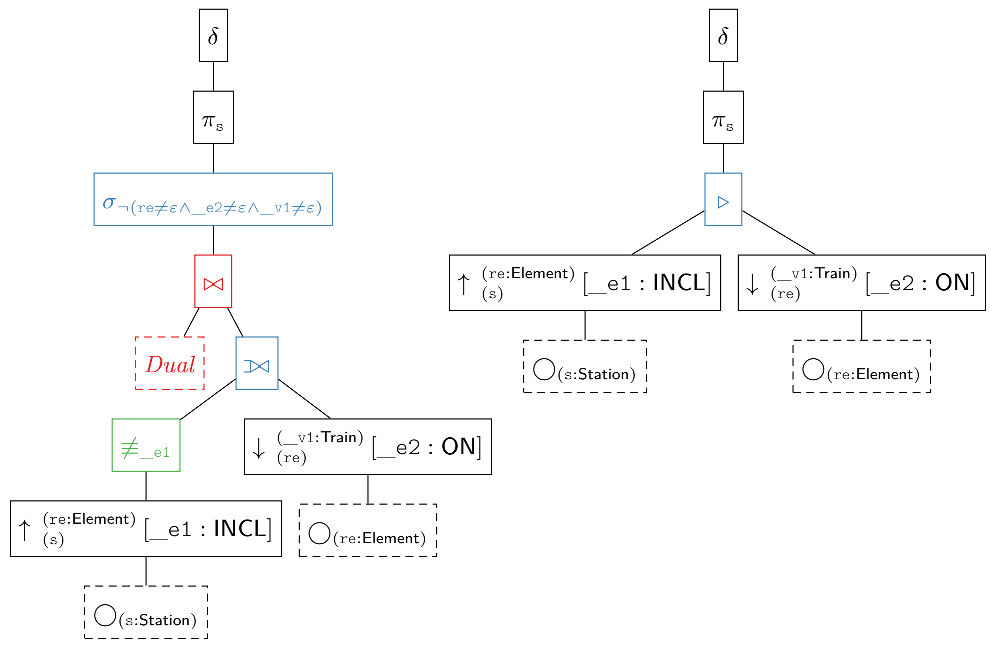
5.3 Identifying Antijoin Operators

```
1 // monitoring objective finds stations that have at least one free track
2 // available
3 MATCH (s:Station)-[:INCL]->(re:Element)
4 WHERE NOT (re)<-[:ON]-(:Train)
5 RETURN DISTINCT s
```

List. 5.3: Station with free track.

The query in List. 5.3 uses negative pattern match on line 4 to express that track element `re` does not have a train on it. This is essentially an antijoin operation. In order to keep compiler simple, the query is compiled in the raw form to the left outer join of the two pattern matches and a negated selection stating that edge and vertex variables of the pattern condition are all non-null ($\neq \varepsilon$). We highlighted the corresponding operator nodes with blue boxes in the raw compiled form of this query, shown in Fig. 6(a). It is transformed by an other VIATRA rule to the antijoin operator, also highlighted using blue in Fig. 6(b).

Simplification of this query again shows the removal of an unused join (highlighted with red). The green box in Fig. 6(a) shows the all-different operator which states that the listed edge variables match unique edges. This is specified by openCypher's edge uniqueness semantics. As one edge is always unique, we added an other transformation rule to remove this operator from the tree.


δ
π_s
$\sigma_{\neg(re \neq \varepsilon \wedge \_e2 \neq \varepsilon \wedge \_v1 \neq \varepsilon)}$
⋈
Dual
$\not\equiv_{\_e1}$
↑ (re:Element) (s) [_e1 : INCL]
↓ (_v1:Train) (re) [_e2 : ON]
○(s:Station)
○(re:Element)
▷

(a) Raw query plan. (b) Simplified query plan.

Fig. 6: Query plans for *Station with free track*.

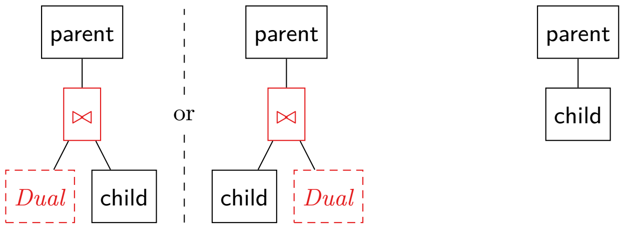
5.4 Formalisation as Graph Transformation Rules

Based on the previous examples, we introduce generic transformation rules for query optimization.

```
pattern parentOperator(op : Operator, parentOp : Operator) {
  UnaryOperator.input(parentOp, op);
} or {
  BinaryOperator.leftInput(parentOp, op);
} or {
  BinaryOperator.rightInput(parentOp, op);
}
```

List. 5.4: Query for determining the parent of an operator.

Removing unnecessary joins. Fig. 7 shows the transformation rule for detecting and removing unnecessary join operators. It looks for *natural join* operators that have a *Dual* operator on one of their inputs and another child operator on their other inputs. If a match is found, it is removed and the child operator is



(a) Left-hand side.   (b) Right-hand side.

Fig. 7: Transformation for removing unnecessary join operators.

```
def changeChildOperator(Operator parentOp, Operator currentOp, Operator
    newOp) {
  switch parentOp {
    UnaryOperator:
      parentOp.input = newOp
    BinaryOperator: {
      if (parentOp.getLeftInput.equals(currentOp))
        parentOp.leftInput = newOp
      if (parentOp.getRightInput.equals(currentOp))
        parentOp.rightInput = newOp
    }
  }
}
```

List. 5.5: Change child operator.

connected directly to the parent operator of the removed join operator. There are no restrictions on the arity of the parent, i.e. it can be either a unary operator or a binary operator.

To implement this rule in VIATRA, we first define a rule that allows us to handle the parent operator in a uniform way. The `parentOperator` pattern in List. 5.4 returns the parent operator `parentOp` of operator `op`. The Xtend code for the transformation rule, which replaces a given child operator `currentOp` of a certain parent operator `parentOp` to a new operator `newOp`, is shown in List. 5.5.

The `unnecessaryJoin` pattern in List. 5.6 uses the `parentOperator` rule to find the parent operator of a certain join operator, checks whether there is a DualObjectSource operator on either the left or the right input of the join operator. The VIATRA transformation rule for removing unnecessary joins is shown in List. 5.7.

Introducing antijoins. In order to evaluate negative conditions efficiently, the optimizer tries to introduce *antijoin* operators where possible. Fig. 8 shows the transformation rule for detecting antijoins. The rule looks for *left outer join* operators that:

```
pattern unnecessaryJoin(childOp: Operator, joinOp: JoinOperator, parentOp
    : Operator) {
  find parentOperator(joinOp, parentOp);
  DualObjectSourceOperator(dualOp);
  JoinOperator.leftInput(joinOp, dualOp);
  JoinOperator.rightInput(joinOp, childOp);
} or {
  find parentOperator(joinOp, parentOp);
  DualObjectSourceOperator(dualOp);
  JoinOperator.leftInput(joinOp, childOp);
  JoinOperator.rightInput(joinOp, dualOp);
}
```

List. 5.6: Determine unnecessary joins. The `parentOperator` pattern is defined in List. 5.4.

```
def removeUnnecessaryJoinOperator() {
  createRule()
    .precondition(UnnecessaryJoinMatcher.querySpecification)
    .action [
      changeChildOperator(parentOp, joinOp, otherInputOp)
    ].build
}
```

List. 5.7: Rule for removing unnecessary joins.

– have a *selection* operator as their parent, which defines a condition that is satisfied iff $\neg\,(\mathtt{v}_1 \neq \varepsilon \wedge \ldots \wedge \mathtt{v}_\mathtt{n} \neq \varepsilon)$ and
– $\mathtt{v}_1, \ldots, \mathtt{v}_\mathtt{n}$ are the variables of the right input of the left outer join operator (see Sec. 5.3).

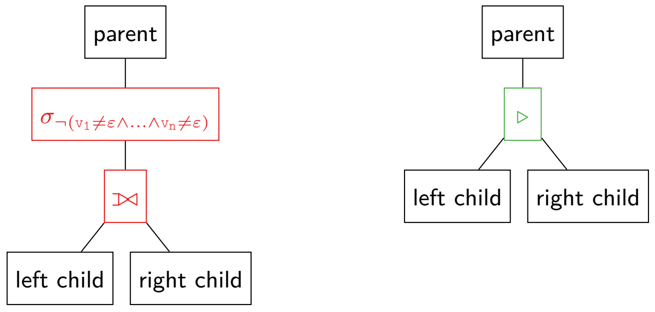
If there is a match, the *left outer join* operator is replaced by a single antijoin operator and the selection operator is removed from the tree.

6 Related Work

6.1 Graph Query Languages

As graph queries are increasingly used in industry, graph query languages are available across different technological spaces. Here, we discuss related query languages and compilers.

Property graphs. The Cypher language was originally designed as the primary query language of the Neo4j graph database system [24,29]. The grammar specification and the language behaviour of openCypher was defined to match those of Neo4j. Consequently, the compiler and query engine of Neo4j form


parent
$\sigma_{\neg(v_1 \neq \varepsilon \wedge \ldots \wedge v_n \neq \varepsilon)}$
⟕
left child
right child
parent
▷
left child
right child

(a) Left-hand side. (b) Right-hand side.

Fig. 8: Transformation for introducing antijoin operators.

the most complete openCypher implementation available, and is dual licensed (GPLv3/AGPLv3 for compatible projects and custom licensing for commercial applications).

The authors of [18] studied the Cypher query language and defined graph-specific relational algebra operators, such as get-vertices and expand-out (Sec. 3.2). While their work focused on optimisation, our work aims to provide a mapping and compilation steps for transforming openCypher to relational graph algebra.

In [19], graph queries were defined in a Cypher-like language and evaluated in the Apache Flink-based Gradoop framework. However, formalisation and compilation of the queries was not discussed in detail.

TinkerPop. The TinkerPop framework aims to define a standard data model for property graphs. For graph queries, it provides the Gremlin Structure API, a low-level programming interface and the Gremlin language, a high-level imperative graph traversal language [31]. The latter is implemented as a Groovy DSL [20].

EMF. Eclipse Modeling Framework (Sec. 3.3) is an object-oriented modelling framework widely used in model-driven engineering. Henshin [3] provides a visual language for defining patterns, while Epsilon [21] and Viatra Query [5] provide high-level declarative (textual) query languages, the Epsilon Pattern Language and the Viatra Query Language (Sec. 3.3), respectively. Viatra Query supports both incremental and search-based queries [9].

RDF. Widely used in semantic technologies, SPARQL is a standardised declarative graph pattern language for querying RDF [47] graphs. SPARQL bears close similarity to Cypher queries, but targets a different data model and requires users to specify the query as triples instead of graph vertices/edges. A formal definition of the language is given in [28]. Apache Jena ARQ [2] and Eclipse RDF4J [10] are open-source compilers and query engines for the SPARQL language.

Comparing graph query engines. The Train Benchmark is a framework for comparing graph query frameworks across different technological spaces, such as property graphs, EMF, RDF and SQL [39].

6.2 Query Compilation in Graph Transformation Systems

The authors of [8] adapted the Rete algorithm originally developed in the domain of production rule systems for pattern matching in a GT engine. The presented solution supported a simple core graph pattern language.

The Fujaba [26] graph transformation tool 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 [34] uses a sophisticated cost model for basic operations and generates the search plan at compile-time by a greedy algorithm.

An algorithm to produce a high-quality (e.g. compact) Rete network from a pattern specification was proposed in [44]. Paper [45] presented an algorithm to define efficient search plans on EMF models. These approaches are used in the eMoflon system [22]. The approach of [46] 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 provides a dynamic, runtime optimization engine, which uses a mix of heuristical and cost-based techniques [16].

The first Viatra prototype, which was capable of generating Prolog code from metamodels and model transformations defined in XMI (XML Metadata Interchange) format, was presented in [42].

The IncQuery-D [38] system is an incremental graph query engine, built on top of the components of the Viatra Query framework [43] (later known as EMF-IncQuery [41]). IncQuery-D reused the query parser and compiler of EMF-IncQuery, but used a different query engine, tailored for scalable distributed query evaluation and operating on RDF data sets.

7 Conclusion and Future Work

In this paper, we presented an approach to design and implement a query engine for the openCypher graph query language. We implemented this approach based on a language workbench built on EMF-based technologies, such as Xcore, Xtext, Xtend and Viatra. The resulting prototype is part of the ingraph project, an openCypher-compatible incremental graph query engine.

In the future, we plan to enhance a query optimizer. A possible approach is to use *search-based optimization techniques using model transformations*, also known as *planning by rewriting* [1]. As our solution already utilizes the Viatra query engine, the optimizer can be based on the Viatra-DSE design-space exploration framework [17] without a significant integration overhead. Another feasible approach is to use Catalyst, a state-of-the-art extensible optimizer framework developed as part of the Apache Spark SQL project [4].

Acknowledgements

The second and third authors of this work were partially supported by the MTA-BME Lendület Research Group on Cyber-Physical Systems. We would like to thank János Maginecz and Dávid Szakállas for their contributions to the relational graph algebra model. We are also grateful to András Vörös and Gábor Bergmann for their suggestions and comments on the draft of this paper.

References

1. J. L. Ambite and C. A. Knoblock. Planning by rewriting. *J. Artif. Intell. Res.*, 15:207–261, 2001.
2. Apache Software Foundation. Apache Jena. `https://jena.apache.org/`.
3. T. Arendt et al. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *MODELS*, pages 121–135, 2010.
4. M. Armbrust et al. Spark SQL: relational data processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.
5. G. Bergmann et al. Incremental evaluation of model queries over EMF models. In *MODELS*, pages 76–90, 2010.
6. B. Bryla and K. Loney. *Oracle Database 12C The Complete Reference*. McGraw-Hill Osborne Media, 1st edition, 2013.
7. Budapest University of Technology and Economics, Department of Measurement and Information Systems. Model-based Demonstrator for Smart and Safe Systems. `https://modes3.inf.mit.bme.hu/`, 2015.
8. H. Bunke, T. Glauser, and T.-H. Tran. An efficient implementation of graph grammars based on the RETE matching algorithm. In *Graph-Grammars and Their Application to Computer Science*, pages 174–189, 1990.
9. M. Búr, Z. Ujhelyi, Á. Horváth, and D. Varró. Local search-based pattern matching features in EMF-IncQuery. In *ICGT*, pages 275–282, 2015.
10. Eclipse Foundation. RDF4J. `http://rdf4j.org/`.
11. Eclipse Foundation. Xtend – Modernized Java. `https://www.eclipse.org/xtend/`.
12. Eclipse Foundation. Xcore, 2017. `http://wiki.eclipse.org/Xcore`.
13. S. Erdweg et al. The state of the art in language workbenches - conclusions from the language workbench challenge. In *SLE*, pages 197–217, 2013.
14. M. Eysholdt and H. Behrens. Xtext: Implement your language faster than the quick and dirty way. In *SIGPLAN, SPLASH/OOPSLA*, pages 307–309, 2010.
15. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems – The complete book*. Pearson Education, 2nd edition, 2009.
16. 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.
17. Á. Hegedüs, Á. Horváth, and D. Varró. A model-driven framework for guided design space exploration. *Autom. Softw. Eng.*, 22(3):399–436, 2015.
18. J. Hölsch and M. Grossniklaus. An algebra and equivalences to transform graph patterns in Neo4j. In *GraphQ at EDBT/ICDT*, 2016.
19. M. Junghanns et al. Cypher-based graph pattern matching in Gradoop. In *GRADES at SIGMOD*, 2017.
20. D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

21. D. S. Kolovos et al. The Epsilon transformation language. In *ICMT*, pages 46–60, 2008.
22. E. Leblebici, A. Anjorin, and A. Schürr. Developing eMoflon with eMoflon. In *ICMT*, pages 138–145, 2014.
23. J. Marton, G. Szárnyas, and D. Varró. Formalising openCypher graph queries in relational algebra. In *ADBIS*, Lecture Notes in Computer Science. Springer, 2017.
24. Neo Technology. Neo4j. `http://neo4j.org/`.
25. Neo Technology. openCypher project. `http://www.opencypher.org/`, 2017.
26. U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In *ICSE*, pages 742–745. ACM, 2000.
27. OrientDB LTD. OrientDB graph-document NoSQL DBMS. `http://www.orientdb.org/`.
28. J. Pérez et al. Semantics and complexity of SPARQL. *ACM TODS*, 34(3), 2009.
29. I. Robinson, J. Webber, and E. Eifrém. *Graph Databases*. O'Reilly Media, 2nd edition, 2015.
30. M. A. Rodriguez. A collectively generated model of the world. In *Collective intelligence: creating a prosperous world at peace*, pages 261–264. 2008.
31. M. A. Rodriguez. The Gremlin graph traversal machine and language (invited talk). In *DBPL*, pages 1–10, 2015.
32. M. A. Rodriguez and P. Neubauer. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology*, 36(6):35–41, 2010.
33. M. A. Rodriguez and P. Neubauer. The graph traversal pattern. In *Graph Data Management: Techniques and Applications*, pages 29–46. 2011.
34. A. Schürr et al. Handbook of graph grammars and computing by graph transformation. pages 487–550. World Scientific Publishing Co., Inc., 1999.
35. A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts, 5th Edition*. McGraw-Hill Book Company, 2005.
36. Sparsity-technologies. Sparksee high-performance graph database. `http://www.sparsity-technologies.com/`.
37. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.
38. G. Szárnyas et al. IncQuery-D: A distributed incremental model query framework in the cloud. In *MODELS*, pages 653–669, 2014.
39. G. Szárnyas et al. The Train Benchmark: Cross-technology performance evaluation of continuous model validation. *Softw. Syst. Model.*, 2017.
40. ThinkAurelius. Titan. `https://github.com/thinkaurelius/titan`.
41. Z. Ujhelyi et al. EMF-IncQuery: An integrated development environment for live model queries. *Sci. Comput. Program.*, 98:80–99, 2015.
42. D. Varró. Automated program generation for and by model transformation systems. In *AGT*, pages 161–174, 2002.
43. D. Varró et al. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Softw. Syst. Model.*, 15(3):609–629, 2016.
44. G. Varró and F. Deckwerth. A rete network construction algorithm for incremental pattern matching. In *ICMT*, pages 125–140, 2013.
45. G. Varró et al. An algorithm for generating model-sensitive search plans for pattern matching on EMF models. *Softw. and Syst. Model.*, 14(2):597–621, 2015.
46. 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.
47. W3C. Resource Description Framework. `https://www.w3.org/RDF/`, 2014.