Vrije Universiteit Amsterdam
Faculty of Sciences, Department of Computer Science

Centrum Wiskunde & Informatica

Georgiana Diana Ciocîrdel
student number 2591443

A G-CORE (Graph Query Language) Interpreter

Master's Thesis in
Parallel and Distributed Computer Systems

Supervisor:
Prof. Dr. Peter Boncz
Vrije Universiteit Amsterdam
Centrum Wiskunde & Informatica

Second reader:
Dr. Alexandru Uta
Vrije Universiteit Amsterdam

August 2018

Abstract

Property graphs are networks of nodes, in which each entity - vertex or edge - can be tagged with labels and assigned properties. Property graph databases store the topology of the connected entities, as well as their attributes. The data can then be queried with dedicated graph query languages (GQLs) that provide subgraph discovery, path discovery or graph analytics through specialized algorithms. G-CORE is a new GQL proposed by the Linked Data Benchmark Council as the sum of the most useful and expressive features in state-of-the-art GQLs for property graphs, with the purpose of guiding the emergence of a standard. G-CORE also brings novelties into the world of GQLs, with queries that return graphs instead of tabular data and by elevating paths to first-class citizens in the graph.

The language definition and semantics of G-CORE are provided in [20]. The goal of this thesis is to verify whether G-CORE is a rich enough language for path property graph databases and whether there can be any improvements we can bring to its design. We achieve this by implementing a G-CORE interpreter that uses SparkSQL and GraphX to leverage the query execution. We cover a subset of the language, that includes pattern matching, graph construction and a particular use-case of path finding and show that we can find a suitable storage model and algebraic primitives to solve G-CORE queries in polynomial time. We also provide a list of issues and ambiguities we have found in the language definition while implementing the interpreter.

Contents

Chapter 1

Introduction

The network-like structure of graphs makes them particularly relevant for modeling the world we live in. As Newman points out in [29], we inherently create *social networks* - in which groups of people interact with each other through friendships, acquaintanceships, business relations, family ties -, disseminate knowledge through *information networks* that relate information items of any type to each other with one famous example being that of a paper citation network -, surround ourselves with *technological networks* - which model the distribution of man-made resources, such as networks of roads, railways, means of communication - and become part of *biological networks* - that are representations of biological systems, such as genes, proteins, the ecosystem.

In practice, graph data is stored in graph databases, which encode data for the graph items, as well as the network topology. Graph query languages can then be used to navigate the network in order to extract or infer information. The Resource Description Framework (RDF) is a popular data format for graph databases, with SPARQL as its effective language standard. As an alternative to RDF, the graph storage can be modeled under the property graph format, which encodes the graph item data inside the items themselves. Numerous property graph query languages have been proposed in academia and some have already proven themselves in the industry. However, none of them has emerged as a standard yet.

The Linked Data Benchmark Council (LDBC) proposed in [20] the new property graph query language G-CORE, designed as the sum of the most common and useful features provided by state-of-the-art property graph query and analytics languages. Among the novelties brought by G-CORE into the field we count queries that return a graph instead of a tabular view over the result, thus enabling query composability, and promoting a new data model, the *path* property graph, by elevating paths to first-class citizens. Currently, there is no implementation of G-CORE in industry or academia. Our goal is to implement the first G-CORE interpreter starting from its definition and formal semantics provided in [20] and assess G-CORE's graph querying model and whether its design is complete, rich and expressive enough for path property graph databases. The challenges will be to find suitable storage primitives for the new data model, as well as designing and implementing the interpretation pipeline with the goal of guaranteeing that the evaluation of queries respects the language semantics. It is not our goal to find the most efficient algorithms for implementing the interpreter or to optimize those that we do propose and we leave this as future work.

1.1 Research questions

Our research questions can therefore be summarized into two categories. First, we aim to assess G-CORE's formal definition:

(Q1) Given its formal definition as presented in [20], are G-CORE semantics rich and expressive enough for path property graph databases?

(Q2) Is G-CORE's formal definition presented in [20] complete? Are there issues with this definition, can improvements be brought to the existing syntax definition and semantics?

(Q3) Is G-CORE indeed a tractable language, as claimed in [20], i.e. can we find suitable tractable algorithms to implement G-CORE's features?

We are also interested in answering more pragmatic questions, closely related to the interpreter implementation:

(Q4) What is a suitable logical representation of the graph data model?

(Q5) What algebraic primitives does G-CORE need beyond normal relational operators?

(Q6) What is a suitable physical representation of the graph data model? What are the trade-offs between various representations?

1.2 Contributions

This thesis outlines our contributions: we present a working prototype system that translates G-CORE to SQL queries and runs them on Apache Spark. We logically model graph data into tables and use DataFrames for the physical representation. Our implementation covers a subset of the language features presented in its design. As a conclusion to our work, we raise a list of questions about less understood G-CORE features and provide ideas on how future work could improve our implementation or enhance the language support.

The remainder of the thesis is structured as follows: Chapter 2 clarifies notions used throughout our work that are related to graph databases and offers a few examples of G-CORE queries that align to the language subset we cover. Related work from industry and academia is presented in Chapter 3. Chapter 4 presents design principles that have guided and influenced the implementation of the interpreter, while Chapters 5, 6 and 7 present in detail the implementation of the modules that comprise the interpreter. In Chapter 8 we outline ideas for improving our solution in order to optimize the query execution and to bring more of G-CORE's features into the proposed prototype and in Chapter 9 we discuss ambiguities we found in G-CORE's language definition. Finally, Chapter 10 provides a summary of our work and answers to the research questions.

Chapter 2

Background

This chapter offers useful background information for understanding the design concepts and implementation details presented further in the thesis. We introduce property graphs and show how G-CORE builds on this data model and extends it to *path* property graphs. We also provide a short introduction to the G-CORE syntax and semantics and showcase some of G-CORE's features through example queries.

2.1 Property graphs

Property graphs are a particular type of graphs that encode network topology, as well as network data stored inside the elements themselves. The items that comprise a property graph are vertices (or nodes) and edges (the relationships between them). A vertex has a unique identifier, a set of incoming edges and a set of outgoing edges, can be tagged with zero or more labels that describe the role or type of that vertex in the graph and can be annotated with a set of key-value pairs called properties or attributes, possibly multi-valued. Edges have direction, a source and a destination vertex and, similar to the nodes, they can be labeled and assigned properties.

Graph databases are defined by Angles et al. in [23] as a particular class of databases, that can store both entity data and information about how the entities connect to each other, which makes them a specifically useful tool for working with graphs, in general, and property graphs, in particular. While in practice the actual physical data model will vary from one system implementation to another, the main feature of the property graph database is that the data instances are organized as graphs with nodes and a connection overlay, with the schema of directed labeled property graphs.

At their core, graphs are data structures that have been studied extensively in the fields of mathematics and computer science. Algorithmic graph theory can be used to analyze and draw useful insights from networks: identifying "central" nodes in a graph, in order to find the most influential and prominent vertices; path analysis to find vertices that can be chained by a set of edges; finding densely connected groups of vertices in the graph, that form so-called "communities"; sub-graph isomorphism to determine whether a structural pattern can be detected within another graph. Lately, standalone graph processing systems, such as Pregel [28], GraphX [27], Apache Giraph [1], have specialized in large-scale graph computations with efficient implementations of iterative algorithms on graphs of billions of vertices and edges.

Graph query languages (GQLs) can be used for navigating the topology of graphs within graph databases and for accessing the data stored along the traversal. The core features of graph query languages are that they support graph pattern matching and path expressions for graph navigation [22]. A graph pattern is nothing more than specifying the shape of a property graph, using variables for nodes and edges, and trying to find a mapping in the graph database for the pattern variables, while preserving the original structure of the graph. Two semantics can be

used for finding this mapping. Under graph homomorphism semantics, a function maps nodes in the database to node variables in the pattern, while preserving the edge structure; the function need not necessarily be injective and multiple variables in the pattern can be mapped to the same database entity, as long as the overall edge structure is preserved. This type of pattern matching is very close to the select-from-where semantics of relational database systems [22]. In the case of graph isomorphism, the mapping function needs to be a structure preserving bijection, in which two distinct query variables can no longer be matched to the same graph entity.

With path expressions the structure of the graph can be navigated in more depth than vertex or edge pattern matching. They can be used to determine whether two nodes can be linked through a sequence of edges (for example, searching for friends-of-friends paths in a social network to suggest new friendships) or to find (weighted) shortest routes between nodes in a graph. In practice, the query space of an arbitrary path can become very large, therefore restrictions on node and edge properties and labels can be used in the query to limit the search space. Regular expressions are a common type of constraints used for defining a language on the labels of the edge sequence along the path.

2.2 The G-CORE language and path property graphs

G-CORE [20] is a new graph query language designed by the Linked Data Benchmark Council[1] for property graph databases. A standard GQL for property graph databases has yet to be defined and adopted, so G-CORE emerges as a collaborative effort between members of the industry and academia interested in synthesizing the desirable aspects of such a standard. To this end, the proposed language was designed with the goal of capturing the core features of existing GQLs, thus being similar in syntax and semantics to PGQL [15], Cypher [11] and Gremlin [3], while still bringing new and interesting characteristics to the table.

G-CORE is a high-level SQL-like language, using an ASCII-art syntax similar to existing GQLs' to express graph patterns. A unique characteristic of G-CORE is that it treats paths (sequences of nodes and labels in the graph) as first-class citizens, which means that paths are part of the data model alongside nodes and edges. Paths are stored in the database, have an identity and can be labeled and assigned properties. G-CORE thus extends the notion of property graphs to that of ***path*** *property graph*s (PPGs). Another novelty in G-CORE is that it is a composable language, meaning that graphs are the input and output of queries. In this way, users can chain commands and create data analysis pipelines over graph databases. We will compare G-CORE with existing GQLs in Chapter 3.

In the remainder of this section we will showcase the main features of G-CORE through examples on the toy graph we introduce in Figure 2.1. The graph represents a social network of a number of characters from the book *A Storm of Swords*, part of the popular series *A Song of Ice and Fire* by George R.R. Martin. The information presented in the graph has been sourced from [25][2], [8] and [12][3] and combined to create a property graph henceforth named "got_graph". The orange circles in Figure 2.1 are nodes tagged with the label Character and each of them has a property "name", for which the value is drawn in a box close to the node. The mauve triangles are nodes labeled House and represent the family or military order the characters belong to. Similar to the characters, the houses have a "house_name", shown in a box next to the node. Using the data from [25] we added a blue edge labeled HAS_MENTION_WITH between two characters that are mentioned in the book no more than 15 words apart. The number of co-occurrences of two characters has been added as the property "#times" to each blue edge. Characters are bound by allegiance to various houses. This relationship has the label

[1] `http://ldbcouncil.org/`

[2] `https://www.macalester.edu/~abeverid/thrones.html`

[3] `https://neo4j.com/blog/graph-of-thrones/#_teaching_graphs`

HAS_ALLEGIANCE_TO and is shown as the pink dotted edges between a Character and one or more House nodes. Finally, houses can fight battles against one another and we represent this with the red edges labeled ATTACKED, between a House node and the other houses it has attacked. The property-key of this type of edge is "battle_name". There are three stored paths in the graph, under the label CATELYN_TO_DROGO. Each path has the property-key "#hops", which represents the number of edges in each path. The graph in Figure 2.1 will be used throughout the rest of this thesis for examples of showcasing various G-CORE features. Figure 2.2 contains an UML diagram describing the graph's data.

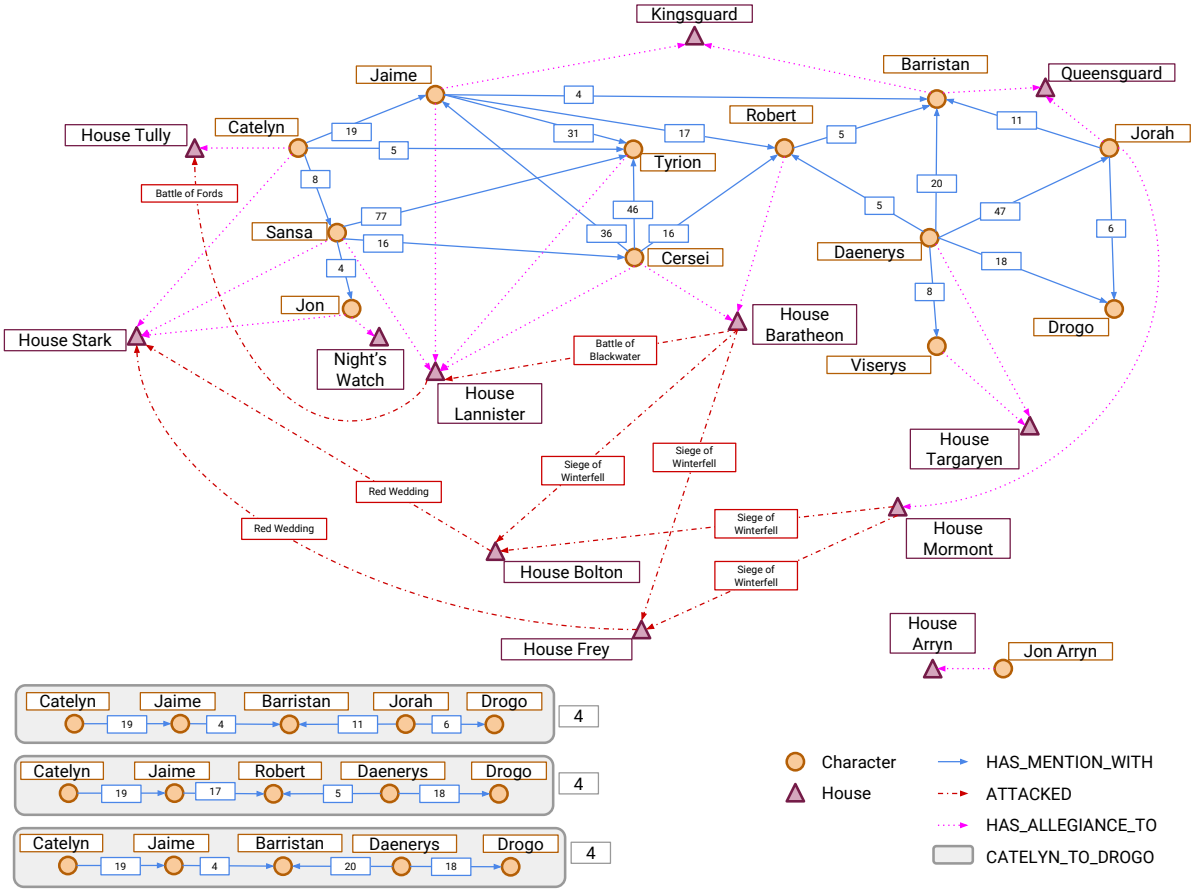

Figure 2.1: An example property graph modeling a restricted view of the universe of the book *A Storm of Swords* by George R.R. Martin. The name of the graph is "got_graph". There are two types of vertices, Character and House, three types of edges, HAS_MENTION_WITH, HAS_ALLEGIANCE_TO and ATTACKED, and one path type, CATELYN_TO_DROGO. The Characters have a property "name" and the Houses a property "house_name". Each edge HAS_MENTION_WITH has a property "#times" and each edge ATTACKED has a property "battle_name". The stored paths have the property "#hops".

2.2.1 A short introduction to G-CORE syntax and semantics

A G-CORE query has clauses, similar to a SQL query, that are chained together, creating variable contexts and passing the results between each other. G-CORE, as presented in [20] is a query language that exclusively supports data manipulation and does not implement the data definition, update or removal operators. Even though a query returns a graph, and the labels and properties of the graph elements can be updated in the result, this represents solely a view

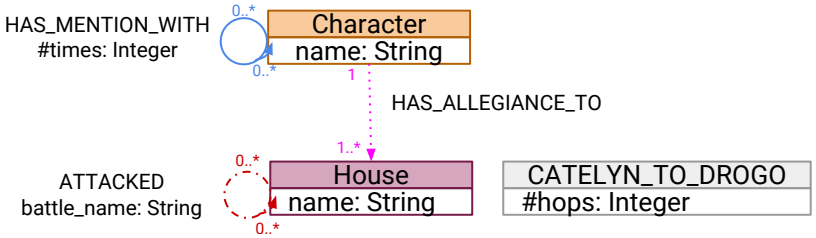

Figure 2.2: Schema of the dataset used to build the graph in Figure 2.1.

returned to the user and is not backed up by changes in the database.

The three basic clauses in G-CORE are the mandatory `MATCH` and `CONSTRUCT` and the optional `PATH` clause. In the listing below we show the basic query structure, using `<token_name>` for tokens. Note that the `PATH`, `WHEN`, `ON`, `WHERE` and `OPTIONAL` clauses can be excluded from a syntactically correct query.

```
PATH <path_macro>, PATH <path_macro>, ...
CONSTRUCT
  <construct_pattern> WHEN <condition>,
  <construct_pattern> WHEN <condition>,
  ...
MATCH
  <match_pattern> ON <graph_query_or_graph_name>,
  <match_pattern> ON <graph_query_or_graph_name>,
  ...
  WHERE <condition>
  OPTIONAL <graph_pattern> WHERE <condition>
  OPTIONAL <graph_pattern> WHERE <condition>
  ...
```

Listing 2.1: Basic structure of a graph query in G-CORE.

The `MATCH` clause is used to specify a pattern to be searched within the graph database. A mapping is created between pattern variables and elements of stored graphs using homomorphic semantics. We call this mapping a *binding table*. Conceptually, the binding table will contain a column for each variable used in the `MATCH` clause and the values will be the corresponding objects of the database. Restrictions can be applied to the elements of the pattern to limit the search space to nodes, edges or stored paths with certain labels. `MATCH` can also be used to fire up a path discovery process within the graph. The `WHERE` clause applies extra conditions on the pattern, by filtering potential results based on property predicates or existential sub-queries. Matching can be performed on multiple graphs at the same time. The `OPTIONAL` sub-clause can be used with left-outer-join semantics to enrich the result with more information from the optional pattern.

The binding table produced by the `MATCH` clause is then used in `CONSTRUCT` to create a new graph. `CONSTRUCT` patterns can re-use variable names from the binding table - in which case the identity of the matched item needs to be preserved -, or can introduce new variables - in which case new items are created by respecting the shape of the binding table. For example, if we matched all the Character nodes in the graph in Figure 2.1 under the variable name `ch`, then constructing a new graph with the node variable `ch` will simply mean reusing the data already mapped to `ch`. In contrast, constructing a new graph with the node variable `foo` (any random name different from `ch`) means that we will need to create additional nodes for the result, but we will create as many new nodes, as have been matched for `ch`. The graph built by a `CONSTRUCT` clause is the union of all the sub-graphs built by each pattern in the clause. `CONSTRUCT` uses implicit grouping of the matched variables to create unique nodes, edges and paths. Additionally, explicit `GROUP`ing can be used by hand for custom aggregation of the binding table.

PATH can be used to define complex patterns for path finding. Remember that path patterns can also be specified in the MATCH clause. However, these patterns can only use simple edge labels to define their structure. For example, in a network of roads (edges) linking towns (nodes), we could match towns that connect through any number of roads. Using PATH we can define a new path structure: roads that are exclusively highways (a road property) and there is a metropolis (node) between each two such roads. We then alias this structure with a macro, say "superRoads" and then try to match towns that are linked by any number of superRoads.

Useful path finding features have been incorporated into G-CORE, such as shortest path finding, all paths finding, reachability tests, weighted paths. Flexible Kleene* expressions can be used to specify desired path structures.

G-CORE supports the following notations for expressing the graph entities comprising a graph pattern:

Nodes `()`, `(v)`
: The shape of a node. The node variable can be left unnamed, or can be given a name - `v`, in our case.

Edges `(a)-[e]->(b)`, `(a)-[]->(b)`, `(a)-->(b)`, `(a)->(b)`
: Relationships between nodes are described with an arrow, either oriented from the source node to the destination node, bidirectional, or undirected. The supported edge orientations are: `->`, `<-`, `<->` (bidirectional) and `-` (undirected). The edges, exactly as nodes, can be named - `e`, in the above - or be left unnamed. We use square brackets to denote the shape of an edge, but the brackets can also be omitted. Node variable names can be omitted altogether.

Paths `(a)-/p/->(b)`, `(a)-/@p/->(b)`, `(a)-/ /->(b)`, `(a)-/@ /->(b)`, `(a)-/p <:EDGE_LABEL*>/->(b)`
: Paths are denoted between two slash signs and can optionally be bound to a variable (`p`, in our example). For paths in particular, not using a variable makes it a reachability query, whereas a bound path will need to be materialized into a chain of edges and nodes. Stored paths, i.e. paths for which information is stored (or is to be stored) in the database, are denoted with `@`. Endpoint names can be omitted and the path orientation is expressed through arrows, exactly as for edges. Sharp brackets can be used to specify a path structure with Kleene* notation, unions and concatenations of multiple Kleene expressions.

Labels `(a:Foo)`, `()-[e:BAR]->()`, `()-/@p:BAZ/->()`
: To match graph items with certain labels, or to assign new labels in the CONSTRUCT clause, we use the `item_name:LabelName` notation. Complex patterns of labels can express label conjunctions and disjunctions, but these do not make the subject of out thesis, as we shall see later on in Chapter 4.

Properties `WHERE a.employer = 'Foo'`, `MATCH (a {employer = e})`, `CONSTRUCT (a {employer := 'Foo'})`
: To limit the matching space to only those graph items for which a certain property predicate holds, we add the respective predicate in the WHERE sub-clause of the MATCH block. G-CORE supports multi-valued properties. To unroll the values of such properties into multiple bindings on single values, we can use curly brackets and the `=` notation when specifying the shape of the element we want to match. Here, we bind the multi-valued property "employer" to the variable `e` and, when binding data for the node `a`, we will actually bind as many entries for `a` as there are values for its "employer" property, but under the variable name `e`. In the CONSTRUCT clause we can assign new properties to the (new) graph elements using the `property_key := property_value` syntax.

Chaining patterns `(a)-[:LIVES_IN]->(city)<-[:LIVES_IN]-(b), (a)-[:FRIEND]-(b)`
Graph patterns can be linked either in the same longer pattern, or by separating them through commas. Here, we are looking for two node variables, `a` and `b`, that both have an outgoing edge labeled LIVES_IN to the same node `city`, adding the condition, through the second pattern, that both be linked by an edge FRIEND (that can be directed from either `a` to `b`, or from `b` to `a`). Pattern chaining in the `MATCH` clause has inner-join semantics, in that common variables between different patterns will keep those bindings that match in all the patterns they appear in.

Among other graph operators supported by G-CORE we list the graph `UNION`, difference (`MINUS`) and `INTERSECT`ion, which take the expected meaning. As G-CORE is a closed language under the path property graph model, the `GRAPH VIEW <view_name> AS (<query>)` operator can be used to create named views of sub-queries, which can be later used as operands to graph operators or as inputs to the `MATCH` clause in other queries.

Examples in Section 2.2.2 will illustrate more clearly the concepts discussed above. [20] abounds in other practical examples, further explanations and a formal definition of the language. An open-source grammar is also available on Github [7].

2.2.2 Examples

This section presents hands-on examples of G-CORE queries that aim to cover the definitions presented in 2.2.1. Some of the examples are inspired from [12] and [20]. We will start with a very simple query that showcases the extraction of a subgraph from the graph in Figure 2.1:

```
1 CONSTRUCT (c1)-[e]->(c2)
2 MATCH (c1:Character)-[e:HAS_MENTION_WITH]->(c2:Character) ON got_graph
```

Listing 2.2: Extract the subgraph of Characters and HAS_MENTION_WITH edges from the got_graph.

In the above, we are first `MATCH`ing the pattern of a HAS_MENTION_WITH relationship, using the variable name `e`. The edge endpoints are labeled Character and have variable names `c1` and `c2`. We then `CONSTRUCT` a new graph from the matched nodes and edges and return this as a result. The new graph will be a sub-graph of the got_graph and will only contain the Character nodes in got_graph and all the blue edges between them. No other node or edge type is present in the new graph. Note that the Character Jon Arryn will not appear in the resulting graph, because it is neither the source, nor the destination of any HAS_MENTION_WITH relationship.

In G-CORE we can set a graph as the default graph for all the queries, thus eliminating the need of using the `ON` sub-clause. From now on, we set got_graph as the default one in our database.

In Listing 2.3 we showcase multiple G-CORE features. The input graph of the outer query is constructed through a sub-query. Within the sub-query we first `MATCH` Houses `h` that have participated in a battle on either side, by leaving the edge `a` as undirected. The binding table of this `MATCH` is shown in Table 2.1. The header of the table contains each variable used in the sub-query and we use the "name" property of the House nodes and the "battle_name" property of the ATTACKED edges to distinguish between the table objects. The semantics of an undirected edge are that the match is performed both as if the edge was an out-connection and as if it were an in-connection, thus duplicating the edge matching in the binding table. To highlight this, we add the extra column "`h` was" to the table, to represent whether the `h` is the source or the destination vertex, and the column "other endpoint", to represent the node that is at the other end of the matched edge. The two extra columns are not part of the actual binding table.

```
CONSTRUCT got_graph, (h1)-[:ALLY {in_battle := b.name}]->(h2)
MATCH (h1:House)-[w1:WAS_IN]->(b:Battle)<-[w2:WAS_IN]-(h2:House)
  ON (
    CONSTRUCT
      (b GROUP a.battle_name :Battle {name := a.battle_name}),
      (h)-[:WAS_IN {role := "attacker"}]->(b) WHEN (h)-[a]->(),
      (h)-[:WAS_IN {role := "defender"}]->(b) WHEN (h)<-[a]-()
    MATCH (h:House)-[a:ATTACKED]-()
  )
  WHERE w1.role = w2.role AND h1.name != h2.name
```

Listing 2.3: Add an edge labeled ALLY between Houses that have fought on the same side in a battle.

h	a	h was	other endpoint
House Lannister	Battle of Fords	source	House Tully
House Tully	Battle of Fords	destination	House Lannister
House Baratheon	Battle of Blackwater	source	House Lannister
House Lannister	Battle of Blackwater	destination	House Baratheon
House Baratheon	Siege of Winterfell	source	House Bolton
House Bolton	Siege of Winterfell	destination	House Baratheon
House Baratheon	Siege of Winterfell	source	House Frey
House Frey	Siege of Winterfell	destination	House Baratheon
House Bolton	Red Wedding	source	House Stark
House Stark	Red Wedding	destination	House Bolton
House Mormont	Siege of Winterfell	source	House Bolton
House Bolton	Siege of Winterfell	destination	House Mormont
House Mormont	Siege of Winterfell	source	House Frey
House Frey	Siege of Winterfell	destination	House Mormont
House Frey	Red Wedding	source	House Stark
House Stark	Red Wedding	destination	House Frey

Table 2.1: The binding table generated by the inner-query in Listing 2.3, with two additional column that better explain the bindings.

The inner `CONSTRUCT` then creates the nodes `b` by `GROUP`ing the binding table by the a.battle_name key. This operation creates four `b` nodes, as there are four distinct battle names in the table: Battle of Fords, Battle of Blackwater, Siege of Winterfell and the Red Wedding. These nodes receive the label Battle and a property "name", which takes the value of the grouping key. We then proceed to add a relationship labeled WAS_IN between the matched Houses and the Battles these Houses were part of. We use `WHEN` to logically partition the binding table into two categories: Houses that were the source of an ATTACKED edge and Houses that were the destination of an ATTACKED edge. Remember that `CONSTRUCT` uses implicit grouping of the binding table for variables that appear in its header and then reuses the identities of these variables, rather than creating new graph instances. In our case, as the node `h` is a matched variable, we will not create new nodes for it and instead use the ones we have already bound. Moreover, since we have already created the nodes `b`, we will group both by source (`h`) and destination (`b`) when creating the edges. Thus, when adding the new edge WAS_IN to the result, we will create six edges with "role" set to value "attacker" between five existing Houses and the four Battles (one with source Lannister, two with source Baratheon, one from Bolton, one from Mormont and one from Frey) and five edges with "role" set to value "defender" between five Houses and

the four Battles (one from Tully, one from Lannister, one from Bolton, one from Frey and one from Stark).

We show the graph created by the inner query in Figure 2.3. As before, property values are drawn in boxes next to nodes or on top of edges.

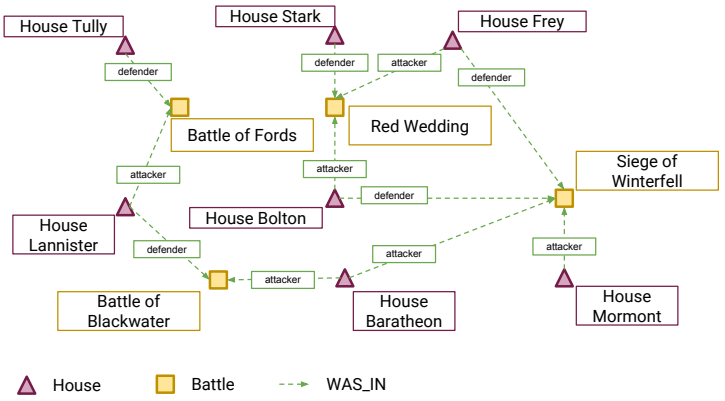

Figure 2.3: Houses and their roles in Battles. The result of the sub-query in Listing 2.3

In the outer `MATCH` we use a chained pattern to search for two Houses `h1` and `h2` that were in the same Battle with an additional condition in the `WHERE` clause that they have had the same role - either both have been attackers or both have been defenders. The chained pattern has inner-join semantics with `b` being the join key. Because of the join, we will generate in the binding table all possible pairs of Houses that have participated on the same side in the same battle.

Finally, in the outer `CONSTRUCT` we use a short-hand for the graph `UNION` by specifying the name of the base graph as a construction argument. We add an edge between `h1` and `h2` if their role in battle was the same, label it ALLY and add to it the property in_battle, which takes as value the name of the battle in which the Houses have been allies. The graph union will enrich the got_graph with the new ALLY edges.

Path finding can be expressed as easily with G-CORE. For example, the stored paths in the got_graph could have been added with the query in Listing 2.4:

```
1 CONSTRUCT got_graph, (c)-/@p :CATELYN_TO_DROGO {#hops := cst}/->(d)
2 MATCH (c:Character)-/3 SHORTEST p <:HAS_MENTION_WITH*> COST cst/-(d:
    Character)
3 WHERE c.name = 'Catelyn' AND d.name = 'Drogo'
```

Listing 2.4: Compute and add at most three shortest paths between the Characters Catelyn and Drogo navigating only HAS_MENTION_WITH edges to the got_graph, labeled CATELYN_TO_DROGO.

In this query, we first search for at most three shortest paths starting from the Character Catelyn to the Character Drogo, going over HAS_MENTION_WITH edges, ignoring their direction. We bind the variable `c` to the cost of the each generated path. When searching for paths in the `MATCH` clause, the cost defaults to the number of edges along the path. For the graph in Figure 2.1 we obtain the following three paths of cost four (five nodes and four edges): Catelyn-Jaime-Barristan-Jorah-Drogo, Catelyn-Jaime-Robert-Daenerys-Drogo, Catelyn-Jaime-Barristan-Daenerys-Drogo. As paths are first-class citizens in G-CORE, we store the result back in the got_graph under the label CATELYN_TO_DROGO. This allows us to later interrogate the database for the known shortest path between the two Characters:

```
CONSTRUCT (c)-/p/->(d)
MATCH (c)-/@p:CATELYN_TO_DROGO/->(d)
```

Listing 2.5: Query the got_graph for paths labeled CATELYN_TO_DROGO.

To search for the weighted shortest path between Catelyn and Drogo we will use the PATH clause, which allows to specify a cost for the path pattern in the clause:

```
PATH wHasMention = ()-[e:HAS_MENTION_WITH]->()
  COST e.#times
CONSTRUCT got_graph, (c)-/@p :W_CATELYN_TO_DROGO/->(d)
MATCH (c:Character)-/p <~wHasMention*>/-(d:Character)
WHERE c.name = 'Catelyn' AND d.name = 'Drogo'
```

Listing 2.6: Same as the query in Listing 2.4, only this time use the property #times of each traversed edge as its cost.

Here we create a path pattern that traverses HAS_MENTION_WITH edges and specify that the hop cost be the edge's property #times, i.e. the number of times a Character has a mention with the Character of the other endpoint of the edge. Then, in the MATCH clause we use this pattern to search for the weighted shortest path between Catelyn and Drogo and add the result to the got_graph under the label W_CATELYN_TO_DROGO. The path that includes Characters with the least co-occurrences in the book is Catelyn-Jaime-Barristan-Jorah-Drogo, with the cost $40 = 19 + 4 + 11 + 6$.

The path pattern can be more complex. For example, we could run the same query, but with the additional constraint that the Character Barristan not be part of the path:

```
PATH wHasMention = (b1)-[e:HAS_MENTION_WITH]->(b2)
  WHERE b1.name != 'Barristan' AND b2.name != 'Barristan'
  COST e.#times
CONSTRUCT got_graph, (c)-/@p :W_CATELYN_TO_DROGO/->(d)
MATCH (c:Character)-/p <~wHasMention*>/-(d:Character)
WHERE c.name = 'Catelyn' AND d.name = 'Drogo'
```

Listing 2.7: Same as the query in Listing 2.6, only this time the Character Barristan must not be part of the path.

The result will be the path Catelyn-Jaime-Robert-Daenerys-Drogo, with the cost $59 = 19 + 17 + 5 + 18$.

The final feature we will showcase are the aggregate functions supported by G-CORE. Using the count aggregation in CONSTRUCTion we can, for example, compute the degree centrality of each Character, i.e. the total number of edges incident to the node. This measure shows us how many other Characters each Character has co-mentions with.

```
CONSTRUCT got_graph, (c {degree_centrality := COUNT(*)})
MATCH (c:Character)-[:HAS_MENTION_WITH]-(:Character)
```

Listing 2.8: Add to each Character node its degree centrality as a new property.

If we want to compute the weighted degree centrality to find the total number of co-occurrences of each Character in the graph, we can instead use the query:

```
CONSTRUCT got_graph, (c {w_degree_centrality := SUM(e.#times)})
MATCH (c:Character)-[e:HAS_MENTION_WITH]-(:Character)
```

Listing 2.9: Add to each Character node its weighted degree centrality as a new property.

During the CONSTRUCTion of the node c the binding table is aggregated by c's identity. As this operation has group-by semantics, we can use aggregations to create new properties for the constructed node.

Chapter 3

Related Work

Graph query languages have been the focus of extensive study in the past decades, however, a language standard for property graph databases has yet to be adopted. The Resource Description Framework[1] (RDF) represents an alternative to the (path) property graph data model. SPARQL[2] is the effective standard graph query language for RDF graph data, which is directed and labeled. While a popular data model, in the RDF format constant literals are encoded as vertices of the graph, making data analysis much more complicated than for property graphs [34]. In comparison, the (path) property graph models information more naturally, by encoding it as properties to graph objects.

Several graph query languages for property graphs have been designed and implemented in practice, but none has been adopted as a standard yet. In [20] LDBC propose G-CORE not as a standard, but rather as a solution to industry's and academia's desires and needs from a query language for graph databases. G-CORE has been designed starting from relevant features offered by three well-established, state-of-the-art property graph query languages, which we will discuss in the following.

Cypher [26] is a declarative language that emerged as a Neo4j product and has later been picked up by other commercial graph database vendors. The openCypher project [18] aims to collect Cypher's capabilities into an open-source grammar and language specification, thus enabling developers and enterprises to leverage Cypher for their own products.

Cypher is currently standardized by the openCypher project at version 9 [5], therefore this version will be the focus of our comparison. Extensive parallels between the semantics and concepts of G-CORE and Cypher 9 are drawn by [20] or [26]. Among them, we mention first the data model, which is that of a property graph for Cypher, while G-CORE elevates paths to first-class citizens and uses a path property graph (PPG) model. There are also differences in query structure and semantics. While both languages use an ASCII-art syntax for graph patterns, G-CORE is closed under the PPG model and constructs a single graph as the result of a query. In contrast, Cypher queries return the matched data in a tabular form, though the `WITH` clause can be used to chain multiple `MATCH` clauses. In Cypher, `WITH` projects columns from the table returned by the first `MATCH`, which become the driving table for the subsequent `MATCH` clause. Moreover, Cypher assumes an implicit graph for all queries, whereas G-CORE features multi-graph queries under join semantics. Given the output format of a Cypher query, its syntax is closer to SQL than G-CORE's. However, [20] proposes a SQL extension for G-CORE for projecting and using tabular data as input to queries. In terms of path matching, Cypher offers similar capabilities to G-CORE, by supporting path queries as regular expressions.

Neo4j [17] is a graph database system that uses Cypher to query natively stored graphs. As

[1] `https://www.w3.org/RDF/`

[2] `https://www.w3.org/TR/sparql11-query/`

outlined in [26], the Neo4j implementation largely translates the Cypher query into a logical algebraic plan and uses a cost-based approach to convert it into an optimal physical plan. This is then either evaluated under a tuple-at-a-time iterator-based model, or compiled to Java bytecode. As a more mature system than the one we will implement in this thesis, Neo4j offers built-in Cypher procedures for common graph algorithms[3], such as centrality measures, community detection and path finding.

Cypher for Apache Spark [4] (or CAPS) is a new open-source project that uses Apache Spark as backend for Cypher queries. Apache Spark [2] is a widely adopted open-source system for large-scale distributed data processing. Spark offers its users the SparkSQL [24] component, a very powerful framework for expressing relational queries on Spark's resilient distributed datasets (RDDs). SparkSQL uses DataFrames, a relational abstraction of RDDs, and the Catalyst optimizer to optimize DataFrame queries. In CAPS, graph data is strongly typed and stored in a tabular form in DataFrames[4]. Cypher queries over the graphs are parsed into a tree-based representation of Cypher-specific and relational operators. After optimizations, the query plan is translated into operations on DataFrames using the DataFrame Scala API[5]. Key contributions of CAPS to Cypher are that the results of queries are graphs and not tables, thus allowing query composability, exactly as G-CORE. The new system also allows working with multiple graphs, instead of a single global graph.

G-CORE also draws inspiration from Oracle's Property Graph Query Language (PGQL) [34]. The standard considered in [20] is 1.1 [14], therefore we will focus our comparison on this version.

Similar to Cypher, the data model in PGQL is that of the property graph. PGQL queries return the set of bindings that matched the given graph patterns in a tabular form. The motivation behind this format is outlined in [34]: PGQL has been designed to offer the users analysis capabilities over graph topologies, but also over the data stored in vertices and edges (their properties). PGQL's syntax is very close to SQL, i.e. queries have the select-from-where structure, with the graph pattern being specified in the FROM clause. This naturally allows the user to extract, process and analyse the structured information inside the graph in the SELECT clause. Patterns are specified in the same ASCII-art syntax as in G-CORE and Cypher. By default, PGQL uses isomorphic semantics for pattern matching, but homomorphic semantics are also supported when so activated in the query. A particularity of PGQL are the expressive path queries that have also found their way into G-CORE and that can be used in the `PATH` clause to form complex path patterns. PGQL's design [34] shows that queries could also return graphs but the semantics are different from G-CORE's: a PGQL query will construct multiple graphs and add them to the tabular result as a *graph* type; G-CORE, on the other hand, unions all the sub-graphs produced by the construct patterns and returns a single PPG.

PGX.D/Async [31] is a distributed in-memory graph pattern matching engine for PGQL queries. The system compiles a PGQL query into a logical plan, then rewrites it to a distributed query plan. The final step is to create an execution plan from the distributed one, by splitting the PGQL query into separate stages, each responsible for matching one vertex. The system uses asynchronous depth-first traversal in a sequential manner. Changes between stages are called "hops" and happen between adjacent graph nodes based on the query pattern. Each stage adds its findings to an output context and, when hopping accross machines, this contex is passed as a message.

Sevenich et al. present in [33] a graph database system that offers graph data consistency and a framework for efficient graph data analysis, called PGX. The authors use PGQL for pattern

[3] `https://neo4j.com/developer/graph-algorithms/`

[4] `https://s3.amazonaws.com/artifacts.opencypher.org/website/ocim2/slides/13-15+Cypher+for+Apache+Spark.pdf`

[5] `https://spark.apache.org/docs/latest/sql-programming-guide.html`

matching queries, coupled with Green-Marl[6] to express graph algorithms. The motivation behind using two different domain-specific languages is that it is difficult to express complex graph algorithms with declarative languages like PGQL, that only specify what data to retrieve/compute, not how. The authors highlight the need for the existence of an imperative language in the system, that can express how a computation should be performed and that is more suitable for creating graph algorithms. The graph storage is relational and offers ACID properties and is built on top of Oracle RDBMS, Oracle NoSQL and Apache HBase. At runtime, data is loaded from the relational graph storage into the analytical engine and the query is not compiled to SQL, thus renouncing the benefits the database SQL optimizers could offer if the query had been performed entirely in the relational world. The authors argue that graph-specific optimizations can instead be applied with their method.

Gremlin [30] is a graph traversal machine and language born under the Apache TinkerPop project [3]. Gremlin supports graph pattern matching semantics, in which query variables are bound to concrete values within the database, as well as the imperative graph traversal model, in which traversal instructions, called *motif*s, are explicitly provided by the user and then a set of traversers move along the graph according to the instructions and collect the traversed objects of the graph into a resulting set. If we view the graph as a physical machine, then the traversal is a program running on it, while the traversers are different instances of the traversal, with their own program counter, registers and data references [6]. The Gremlin traversal machine is essentially an automaton. Traversals can be used to express complex path queries, with cycles, branches and repetitions, being more expressive than the regular path queries.

Gremlin queries are compiled to Gremlin traversals, optimized and then run on the Gremlin traversal machine. The traversal machine can execute the queries on a single machine, as well as in a distributed cluster. TinkerPop implements the Gremlin traversal machine as a virtual machine running inside the JVM. This allowed for numerous Gremlin bindings into programming languages that can be run on the JVM, such as Java, Scala, Ruby, etc. Any graph system can be TinkerPop-enabled, i.e. add support for the Gremlin traversal machine. Through TinkerPop's Gremlin compiler, vendors can register specific traversal optimizations that leverage their respective data model and underlying execution system. Noteworthy Gremlin implementations available in TinkerPop3 are Neo4j-Gremlin[7], SparkGraphComputer[8] or GiraphGraphComputer[9].

In a graph database, graph data can be stored either in a relational model, or under a native graph representation. GraphFrames [9] are an Apache Spark API that combine the relational functionality of DataFrames for representing graphs, with GraphX's powerful graph analytical capabilities. A GraphFrame uses two DataFrames to abstract a graph: one for its vertices and one for its edges. The data model is that of a property graph, where object properties are called "attributes" and stored as DataFrame columns. Each graph object has a unique identifier, which serves the same role as of a primary key, and the edge table contains two additional columns to store the source and destination identifiers of their endpoints - these can be thought of as foreign keys.

GraphFrames support motif finding, a process of graph pattern matching in which graph patterns are described with the same ASCII-art syntax as we have seen before. Join semantics are used to match edges. GraphFrames, however, lack the pattern matching expressivity we have seen in the languages we have previously analyzed. First, in GraphFrames there is no concept of label-based motif finding. Instead, vertices and edges can be attached their label as an extra attribute, and the label condition can be applied as a filtering clause. Explicit labels

[6] `https://github.com/stanford-ppl/Green-Marl`
[7] `http://tinkerpop.apache.org/docs/3.0.1-incubating/#neo4j-gremlin`
[8] `http://tinkerpop.apache.org/docs/3.0.1-incubating/#sparkgraphcomputer`
[9] `http://tinkerpop.apache.org/docs/3.0.1-incubating/#giraphgraphcomputer`

have the advantage of generating a natural partitioning of graph data and acting as types for the graph objects. Also, multi-graph queries cannot be expressed with GraphFrames, as the pattern matching functionality is a specific method of a graph object. For this, data of multiple graphs would have to be merged into a single GraphFrame before applying the pattern. Path patterns are also less expressive. GraphX provides multi-source shortest path finding to a set of landmark vertices out of the box, but more complex path algorithms need to be explicitly implemented by the programmer. While this can easily be achieved by modifying GraphX's algorithm[10], it lacks the elegance of a declarative syntax for path patterns.

[10] `https://github.com/apache/spark/blob/master/graphx/src/main/scala/org/apache/spark/graphx/lib/ShortestPaths.scala`

Chapter 4

Design

In this chapter we outline the design concepts of our prototype system that translates G-CORE queries into SQL statements and then runs them through Apache Spark's SparkSQL engine. We will henceforth refer to this system as the G-CORE interpreter. We wrote the interpreter in Scala[1], used DataFrames to represent the physical data as tables and ran SQL queries on these tables to extract information needed in the query. Compared to the related work discussed in Section 3 our solution is most similar to the Cypher on Apache Spark (CAPS) project.

The design and implementation of the G-CORE interpreter are heavily influenced by preliminary choices concerning the platform used for running queries, how we store and model graph data and, to some extent, even the language subset we decide to cover, although our work strives to be generic and expressive enough to allow more features to be added in the future. We identify three core decisions that considerably influence the rest of our work. The first two are that we settle for a relational model for the graph database and that we translate G-CORE queries into SQL statements. The motivation behind our decisions are that by using tables to store the graph items and represent their topology, and SQL to access and operate with this data we can leverage the functionality of existing RDBMSs, which will save us time from designing our own native or relational representation. Moreover, by translating to SQL as an intermediary step we provide portability to other platforms. The final decision we take is to not strive for performance from the beginning. Our research questions are listed in Sections 1.1 and can be summarized into the main goal of verifying G-CORE's definition through an actual implementation. We leave aside specific optimizations of our interpreter, and instead rely on what the platform we will use for storage and execution can provide. Improvements to our project will be the subject of future work.

Table 4.1 presents the language subset that is covered by our implementation. We list available G-CORE features presented in [20] or available through G-CORE's open-source grammar [7] and highlight how our interpreter deals with each of them.

4.1 Our choice of backend

As the execution and storage layer for our interpreter we decided to use a system that is already well supported and has been proven to be effective in practice. To this end, we considered two candidates, Apache Spark [2] and MonetDB [13] and decided to utilize Spark, due to a better familiarity with the framework. The Spark version we use is 2.2.0. From Spark, we also import the SparkSQL module[2], which allows us to work with structured data. SparkSQL supports SQL queries and uses the Catalyst optimizer to improve performance of query plans [24].

[1] `https://docs.scala-lang.org/`

[2] `https://spark.apache.org/docs/latest/sql-programming-guide.html`

		coverage	notes
	graph view, union, minus, intersection		♠
	path clause		♠
	multi-valued properties, multi-labeled graph items		♣
	SQL extension (tabular queries)		
	data types: integer, string, boolean	□, ◊, △	
	data types: date, time, timestamp	□	
MATCH	graph location: default graphs, named graphs	□, ◊	
	graph location: graph query	□	♡
	vertices, edges, stored or virtual paths	□, ◊, △	
	chained patterns	□, ◊, △	
	labeled vertices, edges, stored paths	□, ◊, △	
	label disjunction, conjunction	□	♡
	property unrolling or aliasing	□	♡
	all stored paths	□, ◊, △	
	shortest, k-(disjunct) shortest stored paths	□	♡[2]
	cost of stored paths	□, ◊, △	
	path expressions on stored paths	□	♡[2]
	shortest virtual path	□[1,2], ◊[1,2], △[1,2]	
	all, k-(disjunct) shortest virtual paths	□[2]	♡[2]
	(weighted) cost of virtual paths	□[2], ◊[2], △[2]	
	path expressions on virtual paths: simple kleene-star	□[2], ◊[2], △[2]	
	path expressions on virtual paths: kleene-star with bounds, union or concatenation, macro	□[2]	♡[2]
	WHERE clause: subset of expressions	□, ◊, △	
CONSTRUCT	set and remove clause	□, ◊, △	
	WHEN clause: subset of expressions	□, ◊, △	
	vertices, edges	□, ◊, △	
	path	□	
	chained patterns in a basic construct	□, ◊, △	
	multiple basic constructs	□, ◊, △	
	copy pattern	□	
	labels and properties added as part of the pattern	□, ◊, △	
	group declaration	□, ◊, △	
	incoming, outgoing edge	□, ◊, △	
	undirected or bidirected edge or path	□	♡
expressions	unary: minus, not	□, ◊, △	
	aggregates: count, collect, min, max, sum, avg, group concat	□, ◊, △	
	arithmetic: mul, div, mod, add, sub	□, ◊, △	
	mathematical: power		
	conditional: (n)eq, gt(e), lt(e)	□, ◊, △	
	logical: and, or	□, ◊, △	
	list operators	□, ◊, △	
	predicates: is (not) null	□, ◊, △	
	existential sub-clause	□, ◊, △	
	function parameters		
	case statements		
	cast		

Table 4.1: G-CORE features covered in our work. Legend: □ parsed, ◊ translated to relational operators, △ translated to SQL, ♣ not covered by data model, ♠ syntactic exception, ♡ unsupported operation exception, [1] incomplete, [2] experimental branch at time of writing.

SparkSQL operates on Datasets, a strongly typed immutable collection of distributed data built on top of Spark's RDDs. Datasets can be transformed in parallel through functional and relational operators. DataFrames represent untyped views over Datasets of generic tuples. In fact, in Scala, the DataFrame is exactly a type alias of a Dataset of rows[3]. With DataFrames, data can be organized into tables of named columns, which perfectly fits one of our core decisions, namely to use a relational model for our data representation. To manipulate Datasets and DataFrames, we can either use the Dataset Scala API[4], or directly SQL statements. This aligns with our desire to translate G-CORE into SQL.

There are, of course, differences between using Datasets versus DataFrames, or the Scala API versus SQL. Compile-time type safety favors certain semantic validations of the query with Datasets when using the API, but not with DataFrames, such as column names and types (columns are typed objects with Datasets). This can increase the runtime errors of DataFrames, even when using languages with static type checking, such as Scala. Another example is that syntax errors will be detected at compile time for both Datasets and DataFrames when using the API, but a SQL query will only be analyzed syntactically and semantically at runtime. However, the common denominator between the two pairs of primitives (Datasets/DataFrames, Scala API/SQL) remains that the Catalyst optimizer will power the transformation phases of the query.

Besides DataFrames, we also considered Spark's GraphFrames [9], which we discussed in detail in Chapter 3. GraphFrames operate on top of DataFrames, but, as we have already highlighted, they are not necessarily an appropriate tool for our use case. Instead of increasing the complexity of our work by trying to modify the open-source GraphFrame code to suit our needs (even if only on a private branch), we decided to use DataFrames directly and implement the entire translation to SQL by ourselves.

4.2 The path property graph data model

This section introduces formal notations for the path property graph data model, that will be used throughout the thesis. We closely follow the G-CORE and Cypher formalisms presented in [20] and [26], respectively.

Let $\mathcal{N}$ be a set of node identifiers, $\mathcal{E}$ a set of edge identifiers and $\mathcal{P}$ a set of path identifiers, with $\mathcal{N}$, $\mathcal{E}$ and $\mathcal{P}$ countably infinite and pairwise disjoint. We denote members of the set as n, e and p, respectively. We assume that we can never run out of identifiers in the system. Further, let $\mathcal{L}$ be a countably infinite set of label names, $\mathcal{K}$ be a countably infinite set of property keys and $\mathcal{V}$ a countably infinite set of literals. We use l to denote a label, k to denote a property key and v to denote a literal (value).

Intuitively, identifiers are values. Our prototype covers three data types supported by G-CORE: integer numbers in $\mathbb{Z}$, finite strings over a finite alphabet Σ and the booleans `true` and `false`. Numbers, strings and booleans are values. `null` is a special value signifying the lack of information. In G-CORE, literals can be manipulated through functions (for example, strings can be concatenated or arithmetic expressions can be applied on numbers).

Here and throughout the rest of the thesis, we use the notations $[v_1, v_2, ..., v_m]$ for lists and $\{k_1 \rightarrow v_1, k_2 \rightarrow v_2, ..., k_m \rightarrow v_m, \}$ for maps (dictionaries), where k_j are keys and v_j are the associated values. Maps can be iterated key-by-key and if m is a map, then $m(k_j)$ is an access to v_j, the value associated with the key k_j. Tuples are represented as $(v_1, v_2, ..., v_m)$ and to extract the ith value of a tuple we can use the Scala-like notation $(v_1, v_2, ..., v_m)$._i, where $i \leq m$.

[3] `https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.package@DataFrame=org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]`

[4] `https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset`

As in [20] we use the notation $FSET(X)$ for the set of all finite subsets of X and $FLIST(X)$ for the set of all finite lists that can be obtained from the elements of X. With this in mind, we define the path property graph model exactly as in [20], as the tuple $G = (N, E, P, \rho, \delta, \lambda, \sigma)$, where:

- N is a finite subset of $\mathcal{N}$, E is a finite subset of $\mathcal{E}$ and P is a finite subset of $\mathcal{P}$, with N, E, P being pairwise disjoint. They represent the graph's nodes, edges and paths, respectively.
- $\rho : E \to (N \times N)$ is a total function that maps each edge identifier to the tuple of source and destination node identifiers. For example, if n_0 is the source of e and n_1 is the destination of e, then $\rho(e) = (n_0, n_1)$.
- $\delta : P \to FLIST(N \cup E)$ is a total function that maps each path identifier to its chain of vertex and edge identifiers. For a path p, $\delta(p) = [n_1, e_1, n_2, e_2, n_3, ..., n_k, e_k, n_{k+1}]$, where $k \geq 0$, $n_i \in N, \forall i \in \{1, 2, .., k\}$ and $e_i \in E$ and $\rho(e_i) = (n_i, n_{i+1}), \forall i \in \{1, 2, .., k\}$.
- $\lambda : (N \cup E \cup P) \to FSET(\mathcal{L})$ is a total function that maps each node, edge and path identifier to a finite, possibly empty, set of labels.
- $\sigma : (N \cup E \cup P) \times K \to FSET(\mathcal{V})$ is a finite partial function that maps a node, edge or path identifier and a property key to a set of values.

To extract path items, we can use $\texttt{nodes}(p)$ and $\texttt{edges}(p)$ to denote the list of nodes and edges of a path p. For example, if $\delta(p) = [n_1, e_1, n_2, e_2, n_3, ..., n_k, e_k, n_{k+1}]$, then $\texttt{nodes}(p) = [n_1, n_2, ..., n_{k+1}]$ and $\texttt{edges}(p) = [e_1, e_2, ..., e_k]$.

4.3 Physical and logical data model

We use the term *graph database* to denote the collection of the available PPGs in our system. Graph data is assumed to be readily available and stored in the form we will describe below. It is not in the scope of the interpreter to validate the data layout. We exclusively support data interrogations - we do not support any data manipulation (such as updates, insertions or deletions) or data definition operations (such as create, drop, import) on the underlying storage. Any manipulation or change on the graphs used in a query will only reflect on the result of that particular query. We are essentially keeping this feature consistent with the outline provided in [20].

We add several changes to the path property graph data model presented in Section 4.2 for the items in our graph database, in order to simplify the interpretation process and storage logic:

- Given that we work in a real, finite system, we will use $L \subset \mathcal{L}$ to denote the finite subset of labels, $K \subset \mathcal{K}$ the finite subset of property keys and $V \subset \mathcal{V}$ the finite subset of literals in the path property graph.
- We change the codomain of λ to L: $\lambda : (N \cup E \cup P) \to L$. In this way, each node, edge and path can only have one label. Further, the images $\lambda(N)$, $\lambda(E)$ and $\lambda(P)$ must be disjoint.
- We change the codomain of σ to V: $\sigma : (N \cup E \cup P) \times K \to V$. In this way, we eliminate the support of multi-valued properties. For each node, edge or path, a certain property key can only take a single value. Because of this, we also disable property aliasing or multi-valued property unrolling in graph patterns.

Next, we model our graph database into a relational storage, in which we represent each label $l \in L$ as a named relation. We use the tuple (l, H, B) to denote this relation: l, a label (which is, after all, nothing more than a string), is the name of the relation; the actual data is stored within a table with header H and body B. The header of the table, H, is a finite set of names and B is a collection of tuples with domain H. Let $\mathcal{D}$ be the relational database that stores the graph data in the form expected by the G-CORE interpreter. If G is a PPG, then $\mathcal{D}(G)$ is the relational representation of the graph and is the tuple $(\nu, R, L_N, L_E, L_P, \theta, \tau)$, where:

- $\nu \in V$ is the name of the graph. We require that every stored graph have a unique name across $\mathcal{D}$.
- R is the set of all named relations that store the graph data. Each element of R is a tuple of the form (l, H, B), where $H \subseteq K$. The notation has the previous explained meaning.
- L_N, L_E and L_P are three, possibly empty, disjoint sets of labels, such that $L_N \cup L_E \cup L_P = L$. They represent the labels of the nodes, edges and paths, respectively, of graph G. The union of the three sets represents the totality of labels in the graph. In the current version of the interpreter, each label must be defined for exactly one graph. However, this can be easily addressed in the future by, for example, prefixing the labels with the graph's name.
- $\theta : (L_N \cup L_E \cup L_P) \to FSET(K)$ is a total function that associates to each label in G a possibly empty sub-set of property keys. Property keys can repeat across labels, however, they will be considered semantically different (for example, in Figure 2.1 if a Character had a property "name" and a House had a property "name", we interpret them as two different property keys, named the same).
- $\tau : (L_E \cup L_P) \to (L_N \times L_N)$ is a total function that maps an edge or path label to a tuple of two node labels, which represent, in this order, the label of the source and destination nodes. The cardinality of the domain is given by the number of edge or path labels available in the graph, so it can also be the empty set when E and P are the empty sets. We are essentially introducing for each graph a constraint on its edge and path labels and require that these only appear between predefined source and destination vertex labels. This will prove useful later on during query analysis. In the current version of the interpreter, this function must be bijective, so we do not allow the mapping of edge or path labels to be multi-valued. This can be easily addressed in the future by changing the data structure that holds this mapping and tweaking the current algorithms.

Depending on the type of label, the header H will be defined as:

- If $l \in L_N$ is a vertex label, $H = \{\text{id}\} \cup \theta(l)$. In other words, for the label of a vertex, we store the vertex identifier along the property keys mapped to that label. The identifier column acts as the primary key of the table.
- If $l \in L_E$ is an edge label, $H = \{\text{id}, \text{src\_id}, \text{dst\_id}\} \cup \theta(l)$. For each record in B with the identifier e, src_id $= \rho(e)$._1 and dst_id $= \rho(e)$._2. In other words, for the label of an edge, we store the edge identifier, the identifiers of the incident vertices and the property keys mapped to that label. While the edge identifier is the primary key of the table, the source and destination ids as foreign keys.
- If $l \in L_P$ is a path label, $H = \{\text{id}, \text{src\_id}, \text{dst\_id}, \text{edge\_seq}\} \cup \theta(l)$. For each record in B with identifier p, src_id$(p) =$ `nodes`(p)._1, dst_id$(p) =$ `nodes`(p)._$k + 1$ and edge_seq $=$ `edges`(p), where k represents the length of the path. In other words, for the label of a path, we store the same columns with the same roles as for an edge, but also a column containing the sequence of edge identifiers along the path.

Mapped on the available Spark primitives we considered in Section 4.1, each table will be represented by a DataFrame. Note that even though we do not support multi-valued properties, DataFrames do support complex data types[5], such as arrays (of which we make use to store a path's edge sequence), maps and structures. This feature can be leveraged in the future to add support for multi-valued properties.

A G-CORE query operates with graphs, rather than with their physical representation described above. Our interpreter needs to be able to run several analysis and rewrite phases on the given query, hence the need to introduce the constraints θ and τ. Moreover, the interpreter needs to be able to infer a table's schema in order to create data domains for the records (the values of the properties). We make the assumption that this is an intrinsic property of the storage and execution layer and with DataFrames this is indeed the case. Hence, θ is built from the underlying data structure, but it could also be provided manually. τ must be explicitly provided by the user, as it cannot be inferred from the underlying data. For example, in our implementation, this information is specified through a configuration file when adding a new graph to the database.

Given the relational representation $\mathcal{D}(G)$ of a graph G, we can reconstruct the graph as follows: using $L_N \cup L_E \cup L_P$ we obtain L, the set of node, edge and path labels in G. For each label in $l \in L_N \cup L_E \cup L_P$, there is a named relation $r \in R$ that stores graph data and each of these relations has a column id. We can thus obtain N, E and P, the identifiers of nodes, edges and paths in G. Further, the information returned by ρ for each edge in the graph is stored in each record in an edge table (reachable through L_E), under the columns src_id and dst_id. We can reconstruct a path from its column edge_seq, which provides the edge identifiers along the path, and the source and destination identifiers stored in separate edge tables. λ will be the relation's label for each identifier in a named relation. Finally, σ is exactly the mapping between the header and the values stored in the body of a relation, for all the relations in R.

To better illustrate these concepts, we show in Table 4.2 how the graph in Figure 2.1 would be represented in our data model.

In the interpreter implementation we make the distinction between the stored graph data and the information we have about it. We use the term *catalog* to denote the global data structure that stores all the graphs' metadata and *graph schema* to denote a light-weight representation of a PPG - it is the same tuple as $\mathcal{D}(G)$, but in which a relation is simply $r = (l, H)$. In other words, the graph schema is the available information about a PPG, modulo the actual vertex, edge or path data: its name, its labels with their associated properties and its label restrictions. Figure 4.1 highlights these concepts. In the Scala implementation, this is translated into a `Catalog` singleton object, which stores a collection of `PathPropertyGraph` objects. Each PPG inherits from the GraphData - with DataFrame fields - and GraphSchema - with the θ and τ mappings and L_N, L_E, L_P fields - classes. Additionally, each PPG also has a name ν.

The catalog offers several primitives: A is the set of all available graph names, with $A \subseteq V$; `register_graph`($\mathcal{D}(G)$) is a function through which metadata about a new graph can be added, when the graph data is added to the database, with its inverse `unregister_graph`(ν), where $\nu \in A$; `set_default_graph(`ν`)` sets a graph as the default graph, given the graph has already been registered in the catalog, with its inverse `reset_default_graph`; `default_graph`, which returns the schema of the default graph, if any graph has been registered as default, or the empty graph otherwise.

[5] `https://spark.apache.org/docs/latest/sql-programming-guide.html#data-types`

Character	
id	name
100	Catelyn
101	Sansa
102	Jon
103	Jaime
104	Tyrion
105	Cersei
106	Robert
107	Barristan
108	Daenerys
109	Viserys
110	Jorah
111	Drogo
112	Jon Arryn

House	
id	house_name
200	Kingsguard
201	Queensguard
202	Night's Watch
203	House Targaryen
204	House Arryn
205	House Mormont
206	House Baratheon
207	House Lannister
208	House Bolton
209	House Frey
210	House Stark
211	House Tully

HAS_ALLEGIANCE_TO		
id	src_id	dst_id
400	100	211
401	100	210
402	101	210
403	101	207
404	102	210
405	102	202
406	103	200
407	103	207
408	104	207
409	105	207
410	105	206
411	106	206
412	107	200
413	107	201
414	110	201
415	110	205
416	108	203
417	109	203
418	112	204

HAS_MENTION_WITH			
id	src_id	dst_id	#times
300	100	103	19
301	100	104	5
302	100	101	8
303	101	102	4
304	101	105	16
305	101	104	77
306	103	107	4
307	103	106	17
308	103	104	31
309	105	103	36
310	105	104	46
311	105	106	16
312	106	107	5
313	108	106	5
314	108	107	20
315	108	110	47
316	108	111	18
317	108	109	8
318	110	107	11
319	110	111	6

ATTACKED			
id	src_id	dst_id	battle_name
500	207	211	Battle of Fords
501	206	207	Battle of Blackwater
502	206	208	Siege of Winterfell
503	206	208	Siege of Winterfell
504	208	210	Red Wedding
505	208	210	Red Wedding
506	205	208	Siege of Winterfell
507	205	208	Siege of Winterfell

W_CATELYN_TO_DROGO				
id	src_id	dst_id	edge_seq	cost
600	100	111	[300, 306, 318, 319]	40

Table 4.2: Logical data representation of the graph in Figure 2.1.

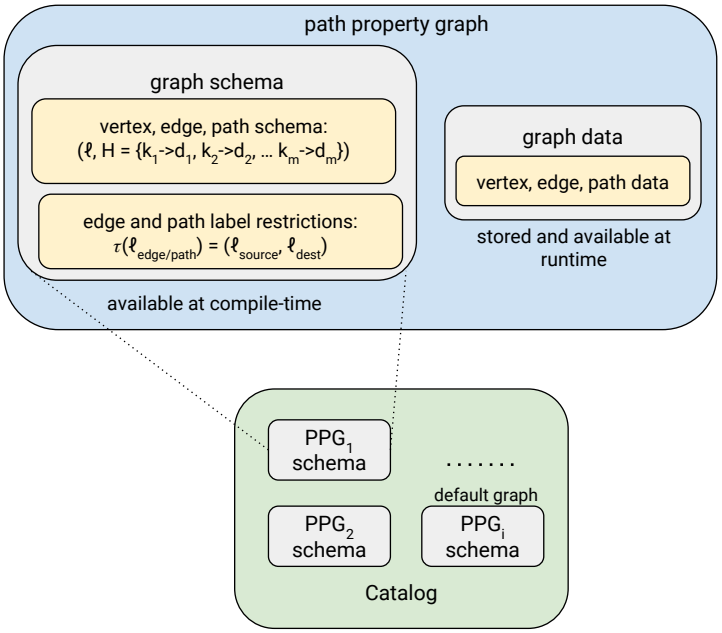

Figure 4.1: Graph metadata is stored in the catalog. The information about a graph is called the graph's schema and contains: a list of tuples (l, H) and a list of label tuples, called label restrictions. l represents a label and H the header of the named relation in which we store data for that label. We store the header as a mapping between property key and the data type of the property values. The label restriction refers to the allowed labels for source and destination vertices of an edge or stored path of a given label. One of the graphs in the catalog can be set as the default graph.

4.4 Overview of the interpretation pipeline

In Figure 4.2 we present an overview of the interpreter and highlight its main components. Our system loosely incorporates canonical compiler stages, but, given the level of maturity of our prototype, is rather simple in its design. A G-CORE query is first run through the parsing module that can detect and emit syntactic exceptions. This stage creates a parse tree and reshapes it in case variable names are missing in the supported clauses. Given the subset of language we support, the root of the parse tree will contain two children, the `MATCH` and the `CONSTRUCT` sub-trees.

The algebraic module then transforms the two sub-trees into an algebraic plan that will initially contain G-CORE-specific operators, but will be sequentially run through rewrite phases that will transform it into a fully relational plan. At this step, the metadata provided by the catalog is used to semantically validate the query or to power certain rewrite or analysis phases. The algebraic module can emit semantic exceptions to signal logical errors in the query, or analysis exceptions to signal coding errors in the interpreter itself. We discuss the difference between the abstracted binding table and the materialized binding table in Section 4.5, but it is important to note that the `MATCH` sub-tree will become one single relational tree, while for the `CONSTRUCT` sub-tree we will need to create as many relational trees as there are variables to be built. The construct rules will need to refer to the binding table created when solving the `MATCH` block. Details about the implementation of this module are given in Sections 6.3 and 6.4.

Once an algebraic plan has been built it can be optimized, however, as mentioned before, this is out of the scope of our project. Instead, the algebraic plan is passed directly to the target module that will transform it into SQL queries. Path finding algorithms are implemented at this level using the GraphX framework [27, 10]. The target module talks directly to the graph

storage, from which it will scan data. Semantic exceptions and runtime exceptions can be raised in this step. We will use one SQL query to evaluate the `MATCH` block and create the binding table and one up to three queries to create each new graph entity. The reason for this is presented in Section 6.4. The target module will finally build and return a path property graph as the result of the query.

Formally, each module in the pipeline is a single-argument function that takes parameters in an input domain I and outputs results in a codomain O. In Scala, each module extends `Function1`[6]. The function uses a chain of tree rewriters on the input. The codomain of one module is the domain of its consecutive module, so the interpretation pipeline becomes the composition of the three stages. If any of the modules throws an exception, the interpretation halts and the subsequent stages are never fired.

We shape the information exchanged by the modules as trees with nodes of a certain data type, also denoted the *tree type*. All tree types in the interpreter extend the base type `TreeNode`, which offers common tree operations, such as traversals or pretty printing. Each node has a possibly empty list of children, of the same type as itself. A node with no children is called a *leaf*. A *tree rewriter* is a two-argument function f which takes as input a tree of type T and a partial function $p : T \to T$, and applies p recursively over the tree nodes, possibly changing its structure. With Scala, each function p extends `PartialFunction`[7]. Depending on how the tree is traversed during the application of p, we distinguish between *top-down rewriters*, which start with the root and descend towards the leaves, and *bottom-up rewriters*, which start with the leaves of the tree and ascend towards the root.

4.5 Binding table abstraction

We have seen in Chapter 2 that the `MATCH` clause is evaluated into a binding table, in which variables in the graph pattern are mapped to the maximal set of values from the database that satisfy the entire `MATCH` block. The binding set can be logically viewed as a table, where each column corresponds to one variable in the clause and the values are complex objects that store not only the identity of a matched object, but also its related properties. We identify the need of materializing the binding table before solving the `CONSTRUCT` clause, as for each new graph entity we will need to group the bindings to produce a PPG and then unite all the results into a single result.

Thus, during the interpretation process we apply two views on the binding table. The distinction between the two is represented in Figure 4.2. First, during the analysis and rewriting phases of the query in the algebraic module, we use the same logical view as presented above, i.e. we view the binding table as a relation over the variables in `MATCH`. This means that the binding table can become the argument of traditional relational operators, such as joins and unions. At this stage we can use the metadata in the catalog for analysis, so explicit graph data is not needed.

The second view is the actual physical view of the binding table, which is produced when the table is materialized before solving `CONSTRUCT` in the target module. As we have seen in Section 4.3, graph data in $\mathcal{D}$ is modeled into named relations of the form (l, H, B), where the header H is comprised of l's property keys and the body B holds the records of the relation. Each variable of the query is labeled or else its label is inferred during one of the rewrite phases of the interpreter. This means that the bindings of all variables will be found in B, where they can be matched to zero or more records, depending on the conditions in the `WHERE` clause. We will also see that the bindings of all the variables in the query will be combined in B through relational operations over their tables, followed by the groupings and other projections needed

[6] `https://www.scala-lang.org/api/2.9.3-RC2/scala/Function1.html`
[7] `https://www.scala-lang.org/api/2.12.1/scala/PartialFunction.html`

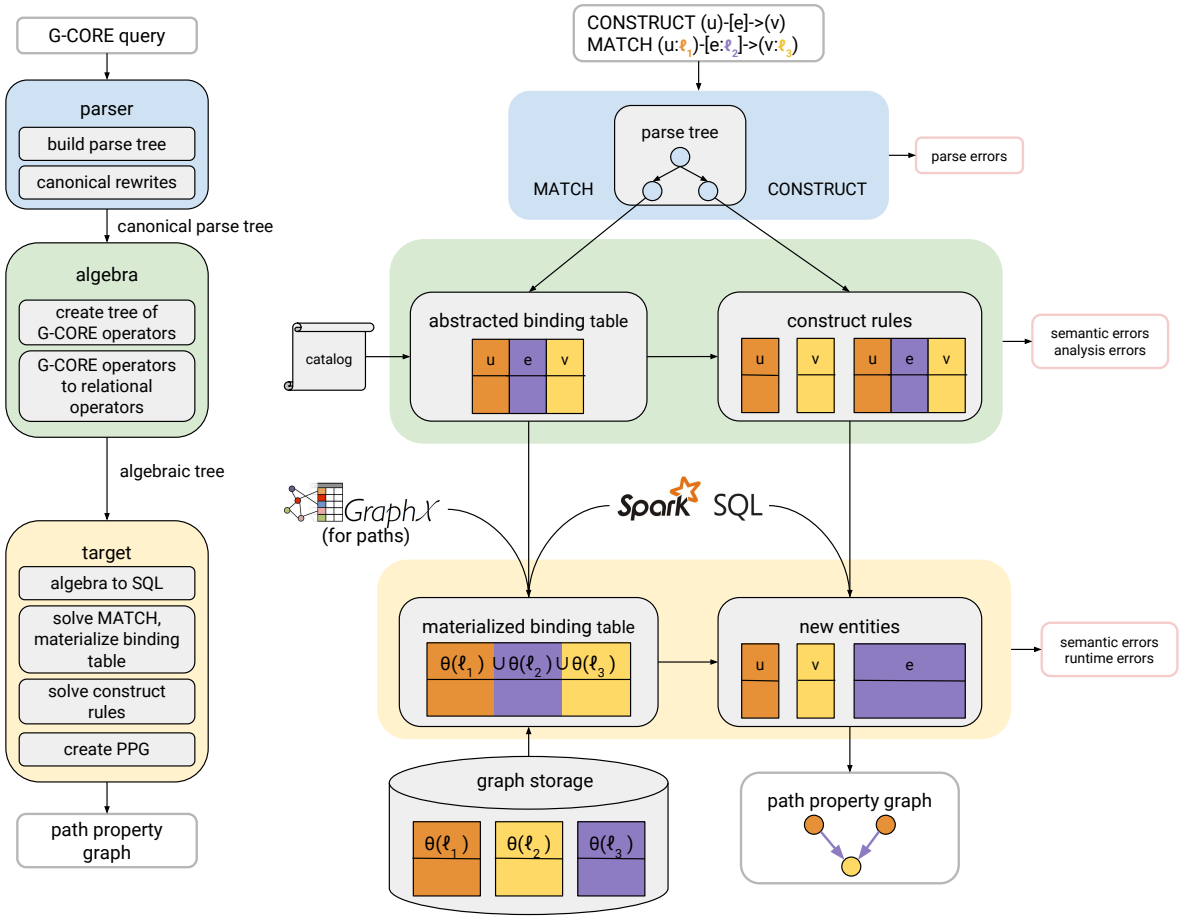

Figure 4.2: An overviews of the interpretation pipeline that translates G-CORE queries into SQL queries.

for construction. So far, these operations are expressed on the abstracted binding table of the `MATCH` clause, but, right before solving the operations of the `CONSTRUCT` clause, they will need to be executed and their result materialized. To this end, the physical view of the binding table will become a single unnamed relation. The header of the binding table will contain all the property keys of the `MATCH` variables and its records will be a combination between the bindings of each variable.

To summarize, during the algebraic rewrites of the query, the binding table is viewed as a relation over the variables of the query, where each value in the table is considered a complex data structure holding the variable's properties. In fact, we are not touching any value in the table at this point. When executing the query, this abstract table is materialized into a relation over the property keys of the labels of each query variable, in which the records will be relational combinations of the bindings of all variables in the query.

Chapter 5

Parser

The interpretation of a G-CORE query starts with parsing the text of the query. This is the first stage shown in Figure 4.2. For this purpose, we used an available open-source G-CORE grammar and parser [7] developed with the Spoofax language workbench [19]. As our project is written in Scala, we take a number of easy steps to import the language specification and use it to parse queries. We first download the project from Github and compile it with Maven[1]. We then zip the generated binary and other sources into a *language component*, which we finally import into our project using the Spoofax Java API[2] seamlessly from Scala. The Spoofax parser produces a syntactic tree using the base type `IStrategoTerm`[3]. We found this type difficult to use in our project, so instead we project this tree into a custom `SpoofaxTreeNode`, a Scala case class. We will use interchangeably the terms *parse tree*, *syntax tree* or *lexical tree* to refer to the result.

The first rewriting of the query is done on the parse tree and has the purpose of canonicalizing it. As we have seen in Chapter 2 the G-CORE syntax accepts unnamed variables both in `MATCH` and in `CONSTRUCT`, for node, edge and path patterns. We address this issue in the canonical rewriter and introduce fresh variable names only for edges and vertices. The semantics of an unnamed path are those of binding its two endpoints only if the destination is reachable from the source, therefore we should not create a binding for paths that have none and thus avoid bringing the path data into the binding table.

parsed pattern	canonical pattern
()	v_i
()-(), ()- -(), ()-[ ]-()	(v_i)-$[e_j]$-(v_k)
()->(), ()- ->(), ()-[ ]->()	(v_i)-$[e_j]$->(v_k)
()<-(), ()<- -(), ()<-[ ]-()	(v_i)<-$[e_j]$-(v_k)
()<->(), ()<- ->(), ()<-[ ]->()	(v_i)<-$[e_j]$->(v_k)

Table 5.1: Canonical rewrite rules of the parse tree. Unnamed variables in `MATCH` and `CONSTRUCT` are given a new name: the prefix is v for vertices and e for nodes, followed by a unique number.

For variable naming, we introduce a number generator $\mathcal{G}$, that generates consecutive natural numbers, starting from 0. If k was the last number generated with $\mathcal{G}$, then at the next request $k+1$ will be emitted. We reshape the parse tree from top to bottom. Depending on whether we encounter an unnamed vertex or an unnamed edge, we will generate a new name for the item, that will start with the prefix v for vertices and e for nodes, followed by the next number yielded

[1] `https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html`

[2] `http://www.metaborg.org/en/latest/source/core/start.html`

[3] `https://github.com/metaborg/mb-rep/blob/master/org.spoofax.terms/src/org/spoofax/interpreter/terms/IStrategoTerm.java`

by $\mathcal{G}$. Table 5.1 shows for which patterns we apply the rewrite rule. We use i, j and k to denote numbers generated by $\mathcal{G}$.

Figure 5.1 presents a query and part of its parse tree. Parse trees can become quite large, so we limit our example to the simplest G-CORE query, `CONSTRUCT () MATCH ()`, in which we try to match an anonymous node on the default graph and then construct an anonymous unbound node from the binding table. None of the variables in the query is bound to a variable name, therefore we introduce two new sub-trees in the syntactic tree to create their names. The result is a canonical parse tree.

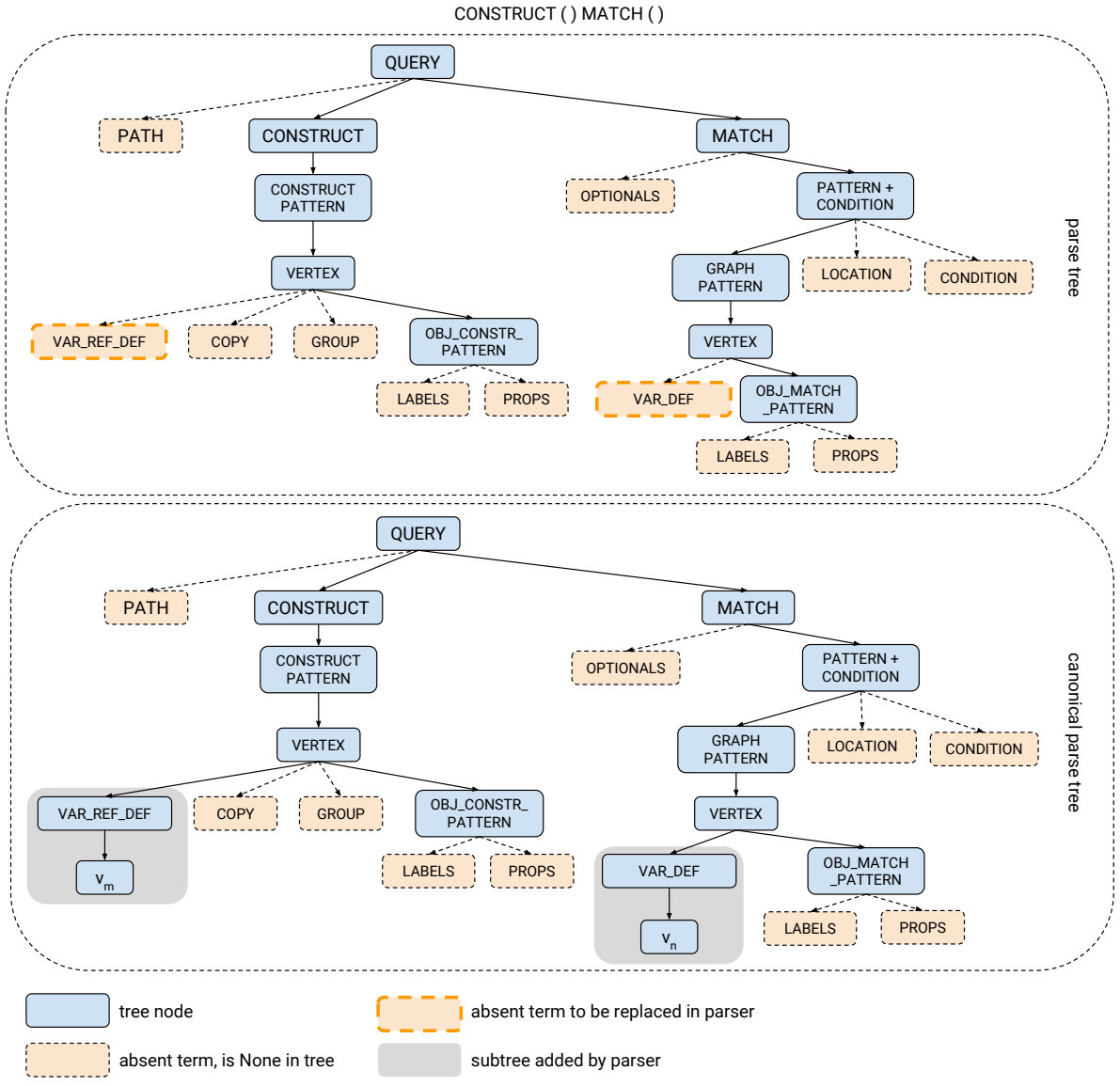

Figure 5.1: Example reshaping of the parse tree for the query `CONSTRUCT () MATCH ()`. Neither the `CONSTRUCT`, nor the `MATCH` variables are named, so the initial parse tree will contain an empty node `None` for each, in place of a variable name. The canonical tree, however, will replace this with the correct sub-tree that introduces the variable names, prefixed with v for vertices.

Chapter 6

Algebra

Once the lexical tree has been rewritten into a canonical form, we can semantically validate the query and then reshape it into a tree of relational operators to create the abstracted binding table and then use it to construct the resulting path property graph. This reshaping is possible because we store graph data into tables, where each table holds the information of a label. We can rewrite the `MATCH` clause into relational operations that build an abstracted binding table. Further, because the binding table is created as a relation over the variables of a query, we can rewrite the `CONSTRUCT` clause into relational operations over this table. It is necessary to go through a number of analysis and rewriting steps to validate and transform the two clauses into fully relational sub-trees. In this chapter we detail each step we took and show how the clauses and sub-clauses of the query can be solved through relational operators.

6.1 The algebraic tree

The parser evaluates a G-CORE query into a raw syntactic tree, which can be cumbersome to validate and process. Therefore, the syntactic tree is iteratively reshaped into an algebraic tree, with the purpose of modeling the query by G-CORE's formal specification [20] and then changing it into relational operations that preserve G-CORE's semantics and produce a PPG.

In the example parse tree in Figure 5.1 we have seen that the root of three is the graph query, which branches into the three G-CORE clauses, `PATH`, `CONSTRUCT` and `MATCH`. If any of the clauses or sub-clauses is missing from the query, it will be replaced by the token None in the parse tree (the orange nodes in Figure 5.1 are, in fact, missing nodes, but we draw them for clarity). In the G-CORE grammar [7] we make the distinction between full *G-CORE queries* and *graph queries*. A G-CORE query is a combination of one or more *graphs* under the operators `UNION`, `INTERSECT` and `MINUS`. Graphs can be specified through their names, or through graph queries. A graph query contains a possibly empty list of `PATH` clauses, exactly one `CONSTRUCT` clause and exactly one `MATCH` clause. Therefore, in Figure 5.1 we are showing a *graph query*.

The evaluation of the `MATCH` sub-tree must result in a maximal set of bindings that satisfy the entire `MATCH` block. We will represent the binding set as a relation in which the header contains every variable in the `MATCH` block and the body contains the bindings for each variable. In the evaluation of the `CONSTRUCT` block we will use the materialized view of the binding table to build a new PPG. Given our relational representation of both stored data and binding table we will make use of several of the extensive relational operators supported by DataFrames. For a quick reference, we define them in Table 6.1, using the unnamed relations, $r = (H, B)$, $r_1 = (H_1, B_1)$ and $r_2 = (H_2, B_2)$, where H_i represents the header of the relation and B_i its body, i.e. the actual records.

The base type of all the operators in the algebraic tree is the `AlgebraTreeNode`. As we have seen in Table 4.1, we exclusively support graph queries, therefore the root of the algebraic tree

Notation	Semantics
$r_1 \bowtie_P r_2$	The two relations are inner-joined based on condition P, a predicate that tests a relation between attributes of r_1 and r_2. This operation will produce all combinations of records in B_1 and B_2 that satisfy the join condition.
$r_1 \mathbin{⟕}_P r_2$	The two relations are left-outer-joined on condition P. This operation will join the records in r_1 and r_2 on their common attributes, but records in B_1 that do not participate in the join will also be preserved in the result.
$r_1 \ltimes_P r_2$	The two relations are left-semi-joined (or semi-joined, for short) on condition P. The semantics are similar to the inner-join, except that the attributes of r_2 do not appear in the result and a matched record in r_1 will only appear once in the result.
$r_1 \times r_2$	The two relations are cross-joined. The operation produces all possible tuples (b_1, b_2), with $b_1 \in B_1$ and $b_2 \in B_2$.
$\Pi_{a_1,a_2,...}(r)$	All the tuples in r's body B are restricted to the set of attributes $\{a_1, a_2, ...\}$, where $a_j \in H$.
$\sigma_P(r)$	Selects those records in B that conform to the predicate P.
$\Gamma_{k_1,k_2...;f_1(a_1),f_2(a_2)...}(r)$	The relation r is aggregated using the combination of attributes $k_1, k_2, ...$ as key. An arbitrary number of aggregation functions f_j are applied on to attributes a_j that are not part of the key.
$r_1 \cup r_2$	The two relations are combined under set union into a new relation. The header of the result will contain all the common attributes in H_1 and H_2, but also attributes that appear in either one of the relation, but not in the other. The body of the result will contain all the records in r_1 and all the records in r_2. If, for a record, a given attribute is not defined in the relation of origin, we bind it to the value `null` in the result.

Table 6.1: Relational operators used throughout this chapter and their meaning.

will be a node Query and, as we do not support the `PATH` clause, its children will be a Match and a Construct node. The types of the algebraic nodes are implemented as Scala case classes. Starting from the `AlgebraTreeNode`, we define *algebraic types*, *expressions* and *operators*, each as an algebraic node type. Algebraic types can be various graph types or items that occur in a match or construct pattern. Vertices, edges, paths, edge and path orientation are all algebraic types, as well as the default graph, a named graph or a graph expressed through a sub-query. The algebraic expressions can be unary or binary expressions, conditional, logical, predicates, and so on, that can be applied on bindings. Labels and property keys are also algebraic expressions. Algebraic operators model the main clauses of a query and also relational statements. For example, Match and Construct are operators, but also Project, GroupBy or Select.

There is also a distinction between G-CORE-specific and relational operators. At first, a parse tree will be mapped to G-CORE-specific node types, which closely follow the tokens and statement structure in the G-CORE grammar [7]. Iterative rewriting steps will combine the G-CORE operators to create or to operate with the binding table, transforming the algebraic tree into a fully relational tree. Sections 6.2, 6.3 and 6.4 provide details abot expressions and G-CORE specific operators that we used in the interpreter, and elaborate the mentioned rewriting phases.

6.2 Expressions

Expressions can be used in a G-CORE query in the `WHERE` and `WHEN` sub-clauses to filter the binding table, or to define objects that will be constructed. In Table 4.1 we showed which expressions are supported by our interpreter. In our implementation, all expressions extend the base class `AlgebraExpression`, but we further split them into different expression types, as we will see below.

Let x be a bound variable, k a property key and l a label. *Basic expressions* are literals (values) and variable references, labels and property keys, property values $x.k$, label predicates $x : l$, label disjunction $\vee l_k = l_1 \vee l_2 \vee ... \vee l_k$ and conjunction of label disjunctions $(\vee l_{k_1}) \wedge (\vee l_{k_2}) \wedge ... \wedge (\vee l_{k_m})$, lists of labels $[l_1, l_2, ...l_k]$ or property keys $[k_1, k_2, ...k_m]$, object patterns in pattern and construct tuples, the aggregation star symbol `*` and the existential sub-query `EXISTS q`. *Unary expressions* are the unary minus, negation `NOT` and the *predicate expressions* `IS NULL` and `IS NOT NULL`. *Aggregate expressions* are a particular type of unary expressions and are one of the functions `collect`, `count`, `min`, `max`, `sum`, `avg`, `group-concat`. All other expressions supported by the interpreter and enumerated in the following are *binary expressions*. *Arithmetic expressions* are the multiplication `*`, division `/`, modulo `%`, addition `+` and subtraction `-`. *Conditional expressions* are the comparison operators for equality `=` and non-equality `!=`, greater than `>` and greater than or equal to `>=`, lower than `<` and lower than or equal to `<=`. *Logical expressions* are `and` and `or`. The power function `pow` is an example of a *mathematical expression*.

As all expressions extend the same base type and are implemented as Scala case classes, they are composable, so they can receive as argument any other expression from the ones enumerated. The rules for evaluating each type of expression are detailed in G-CORE's design [20]. At the algebraic level, we do nothing more besides creating the expression sub-tree. Ideally, semantic checks should be performed on this sub-tree to validate it - for example, argument type compatibility should be implemented. However, this is not available in the implementation at the time of writing.

[20] also mentions that evaluating an existential condition `WHERE <graph_pattern>` is equivalent to evaluating the existential sub-query `WHERE EXISTS (CONSTRUCT () MATCH <graph_pattern >)`. We note that `EXISTS` evaluates to `true` iff the graph constructed in the sub-query has a non-empty set of nodes, and to `false` otherwise. This is in fact very similar in semantics to SQL's EXISTS clause, which can be solved through a semi-join. In the implementation of the interpreter we do not apply the suggested translation. Instead, we wrap the graph pattern into an Exists node and evaluate this pattern into a binding table in the *target* module, followed by a translation to SQL's EXISTS. We show below why this works.

Let φ_1 and φ_2 be two graph patterns, `MATCH` φ_1 `WHERE` φ_2 a G-CORE query with an existential predicate, Ω_1 the binding table produced by the evaluation of φ_1, Ω_2 the binding table produced by the evaluation of φ_2 and Ω the binding table of the entire `MATCH` block. Then $\Omega = \Omega_1 \ltimes_P \Omega_2$, where the condition P is either the equality predicate on common attributes of Ω_1 and Ω_2 if the two patterns φ_1 and φ_2 are correlated (they share variables) or $\Omega_2 \neq \varnothing$ otherwise, where $\varnothing$ is the empty set.

For an easy informal proof, let's first assume that φ_1 and φ_2 are correlated and share a variable x. If x is a vertex and Ω_2 is non-empty (i.e. we did find bindings for the vertex x that satisfied the pattern φ_2) it means that the graph that would have been constructed from Ω_2 would have been non-empty and `WHERE` φ_2 would evaluate to `true` for each binding tuple in Ω_1 in which x has one of the values to which it has been bound in Ω_2. As an example, if x was bound to three vertices with ids $\{1, 2, 3\}$ in Ω_1 and to four vertices with ids $\{2, 3, 4, 5\}$ in Ω_2, in Ω x will retain the ids $\{2, 3\}$. If x is an edge or a path and Ω_2 is non-empty, it follows that there would be nodes in the graph built from Ω_2, because an edge always has two incident nodes and a path is comprised of edges. Therefore, the same logic that applied for a vertex also applies here.

Now, regardless of x's type, if Ω_2 were empty, the graph built from it would be empty, so none of the tuples in Ω_1 should be selected into the result, which is indeed the case with our rewrite rule. Finally, for non-correlated patterns φ_1 and φ_2, the evaluation of the predicate $\Omega_2 \neq \phi$ depends on Ω_2's size (or number of bindings). Using the same logic as before, this means that if φ_2 cannot be satisfied by any binding, Ω_2 will be empty and no tuple from Ω_1 will be selected. Otherwise, whatever the number of bindings in Ω_2, all tuples in Ω_1 will be added to the result.

6.3 The MATCH clause

The role of the `MATCH` clause in a graph query is to bind all the variables used in its block to actual values stored in the graph database, by preserving the structure of the given graph patterns. G-CORE uses homomorphic semantics to match the patterns. Graph patterns can be specified directly after the `MATCH` keyword, in the attached `OPTIONAL` sub-clauses or even in the `WHERE` expression. The evaluation of the entire `MATCH` block must result in the maximal set of bindings that satisfy the entire block.

We initially build the algebraic `MATCH` sub-tree from G-CORE-specific operators that emulate the syntax defined at [7]. We introduce these operators in the current section. We will then translate this sub-tree of G-CORE operators into a tree of relational operators that will keep the evaluation on par with the semantics described in [20]. This section will provide details about each analysis routine and rewrite rule that we used to achieve this purpose. At the end of the entire rewriting pipeline, we will have transformed the `MATCH` clause into a chain of operations that can be run on the tabular representation of the graph data to create the required maximal set of bindings.

6.3.1 Algebraic representation of vertices, edges and paths

Vertices, edges and paths are graph entities used in the graph topologies of the `MATCH` block. We implement the three types Vertex, Edge and Path as case classes that extend the same base class. Formally, we represent each type as a tuple of various items with specific meaning. We will use π to denote an entity pattern.

The pattern of a vertex is the tuple (v, L_v), where v represents the variable used in the query to denote the vertex and L_v is a possibly empty finite set of labels. While the query need not necessarily label its vertices, we restrict the cardinality of L_v to at most one. This restriction is enforced through a semantic check on the vertex tree node and stems from our data model, which exclusively supports graph entities with a single label, and from the supported language set, which disables label disjunction in the queries. Therefore, L_v is either the empty set $\varnothing$, or $L_v = \{l_v\}$. Note that $L_v \subset L_N$[1]. For example, the vertex pattern `(c)` is $\pi_c = (c, \varnothing)$ and the pattern `(c:Character)` is $\pi_c = (c, \{\text{Character}\})$. The pattern `(c:Character|House)`, which matches either Character or House, would generate an unimplemented operation exception.

The pattern of an edge is the tuple $(e, L_e, \pi_L, \pi_R, d)$, where e is the variable used in the query to denote the edge, π_L and π_R are the patterns of the two left and right endpoints, $d \in \{\rightarrow, \leftarrow\}$ is the edge's direction and L_e holds the same meaning as L_v from above, only that it is applied for the edge e. Note that G-CORE also accepts undirected or bidirectional edges, but we currently do not support them, hence the restricted domain for d. For example, the edge pattern `(c)-[e]->(h)` is $\pi_{c \xrightarrow{e} h} = (e, \varnothing, (c, \varnothing), (h, \varnothing), \rightarrow)$ and the edge pattern `(h)<-[e:HAS_ALLEGIANCE_TO]-(c)` is $\pi_{c \xrightarrow{e} h} = (e, \{\text{HAS\_ALLEGIANCE\_TO}\}, (h, \varnothing), (c, \varnothing), \leftarrow)$.

The pattern of a path is the tuple $(p, rt, L_p, q, \pi_L, \pi_R, d, o, c, r)$, where p is the variable name used in the query to denote the path, π_L, π_R and d hold the same meaning as for edge, with

[1] In Section 4.3 we denoted with L_N, L_E and L_P the three disjoint sets of labels of nodes, edges and paths, respectively, in a graph G

the same restrictions and L_p is the same as L_v or L_e, only that it is applied for the path p. rt and o are boolean values - the former encodes whether the path is a reachability test and the latter whether the path is stored - if `false`, the it is virtual and must be computed. q represents the path quantifier and is either `ALL` (meaning that we are searching for all paths between the two nodes) or the tuple $(n, distinct)$, where n is the maximal number of bindings we accept for path p and $distinct$ is a boolean showing whether we are looking for distinct paths or not. r is a regular path expression and can either be omitted from the query, in which case we use the empty set notation $\varnothing$, or a Kleene expression. We only support Kleene-star expressions on a single label, so r can then be the single-element list $\{l_r\}$, where l_r is the edge label. Finally, c is either the variable name that binds the cost of the path, if specified, or the $\varnothing$ notation, otherwise.

In Chapter 5 we noted that the parser will rewrite the syntactic tree into a canonical form, in which it will add fresh variable names for the unnamed vertices and edges. The semantics of an unnamed path are that it is a reachability test. However, to be able work with the path pattern during algebraic rewritings, we need to be able to refer it by name. Therefore, when transforming the syntactic sub-tree of a path into its algebraic representation, if the path is unnamed, we create a new name for it and set rt to `true`. Otherwise, if it was named, rt =`false`.

To showcase path representation, the pattern `(c)-/p/->(d)` is $\pi_{c \overset{\text{P}}{\Rightarrow} d} = (p, \texttt{false}, \varnothing, (1, \texttt{false}), (c, \varnothing), (d, \varnothing), \rightarrow, \texttt{false}, \varnothing, \varnothing)$, `(c)-/@p/->(d)` is $\pi_{c \overset{@_{\text{P}}}{\Longrightarrow} d} = (p, \texttt{false}, \varnothing, (1, \texttt{false}), (c, \varnothing), (d, \varnothing), \rightarrow, \texttt{true}, \varnothing, \varnothing)$ and `(c)-/<:HAS_MENTION_WITH*>/->(d)` is $\pi_{c \overset{@_{\text{P}}}{\Longrightarrow} d} = (p, \texttt{true}, \varnothing, (1, \texttt{false}), (c, \varnothing), (d, \varnothing), \rightarrow, \texttt{false}, \varnothing, \{\text{HAS\_MENTION\_WITH}\})$.

Using the above algebraic representations, variable names will always be defined for each item of a pattern. On each representation we perform semantic checks and, when the combination of tuple elements is not support, we throw an unsupported operation exception. For each vertex, we check that any label used in L_v is part of the graph on which the vertex is matched. The same check is done for edge and path representation, with the additional constraint that they are valid iff their endpoints have passed the sanity check.

6.3.2 MATCH operators

The interpreter translates the entire `MATCH` block, including `OPTIONAL` matches and the `WHERE` sub-clause, into a tree-like representation of G-CORE-specific algebraic operators. The result is the `MATCH` sub-tree in the algebraic tree.

We previously defined object patterns as graph objects with a variable name attached, that need to be bound to values in the database, and denoted them with π. We now define *graph patterns* as per the G-CORE grammar [7] as either a vertex pattern, or a vertex pattern followed by zero or more connection patterns. A connection pattern is either the pattern of an edge, or the pattern of a path, followed by the pattern of the destination vertex. In the previous section we showed how we represent each graph object - vertex, edge or path - in the algebraic tree. Graph patterns are more than simple graph objects, as they can also encode chained patterns. For example, `(a)`, `(a)->(b)`, `(a)->(b)<-(c)`, `(a)-/ /->(b)<-/ /-(c)` are all examples of graph patterns. We will use $\varphi_{\pi_1 \pi_2 ... \pi_n} = [\pi_1, \pi_2, ...\pi_n]$ to denote the algebraic representation of a graph pattern with $n \geq 1$ graph objects. The algebraic form of the graph pattern is an n-ary list: if $n = 1$ the only element of the list can be a vertex, edge or path; if $n > 1$ the list can only contain edges or paths. The order of the elements in the list is important and must be the same as the order in which the graph objects appear in the pattern. In the Scala implementation the GraphPattern is a case class which takes as argument a list of Connection objects, which are either Vertex, Edge or Path objects. For example, the pattern `(a)` becomes $\varphi_a = [(a, \varnothing)]$, `(a)-[e]->(b)` becomes $\varphi_{a \overset{\text{e}}{\rightarrow} b} = [(e, \varnothing, (a, \varnothing), (b, \varnothing), \rightarrow)]$ and `(a)-[e]->(b)-[f]->(c)` becomes $\varphi_{a \overset{\text{e}}{\rightarrow} b \overset{\text{f}}{\rightarrow} c} = [(e, \varnothing, (a, \varnothing), (b, \varnothing), \rightarrow), (f, \varnothing, (b, \varnothing), (c, \varnothing), \rightarrow)]$.

Next, we use the term *match tuple* to denote a graph pattern followed by a *location* - the default graph, a named graph, or a graph sub-query. As shown in Table 4.1 graph sub-queries are parsed, but an unsupported operation exception is thrown when one is encountered. Using syntactic tokens, the match tuple can be expressed as `<match_tuple> ::= <graph_pattern> | <graph_pattern> <location>`. When the location is not provided in the tuple, it becomes the default graph. The match tuple is the first G-CORE-specific operator we introduce in the interpreter and is a tuple $M = (\varphi_{\pi_1\pi_2...\pi_n}, G)$, containing a graph pattern and a graph G, which is either the default graph G_0 or a named graph G_{name}. For example, `(a)->(b) ON got_graph` is the match tuple $M = (\varphi_{a \xrightarrow{e} b}, G_{\text{got\_graph}})$ and `(a)->(b)`, in which we do not specify the graph, is the match tuple $M = (\varphi_{a \xrightarrow{e} b}, G_0)$.

The second G-CORE operator we introduce is the *conditional match* $\dot{M} = ([M_1, M_2, ...M_n], \xi)$. It consists of a list of n match tuples, with $n \geq 1$, and a condition ξ that filters the binding table produced by the combined evaluation of all match tuples. ξ is, in fact, an expression. When the condition is not specified in the query, we fill it in as `true` for uniformity. For example, the query `(a)->(b), (c)` is a conditional match with the condition left unspecified and with two match tuples. It will become $\dot{M} = ([(\varphi_{a \xrightarrow{e} b}, G_0), (\varphi_c, G_0)], \texttt{true})$. The query `(c) WHERE c.name = 'Catelyn'` becomes $\dot{M} = ([(\varphi_c, G_0)], \texttt{c.name = 'Catelyn'})$.

The final G-CORE operator we introduce is the full *match clause* $\overline{M} = (\dot{M}_1, [\dot{M}_2, \dot{M}_3, ...\dot{M}_n])$, which encodes the entire `MATCH` block of a graph query and consists of one conditional match - the non-optional component of the clause - and a possibly-empty list of conditional matches - the `OPTIONAL` sub-clauses. Figure 6.1 presents the breakdown of a match clause into conditional matches and match tuples and Table 6.2 summarizes the patterns and G-CORE operators introduced in this section.

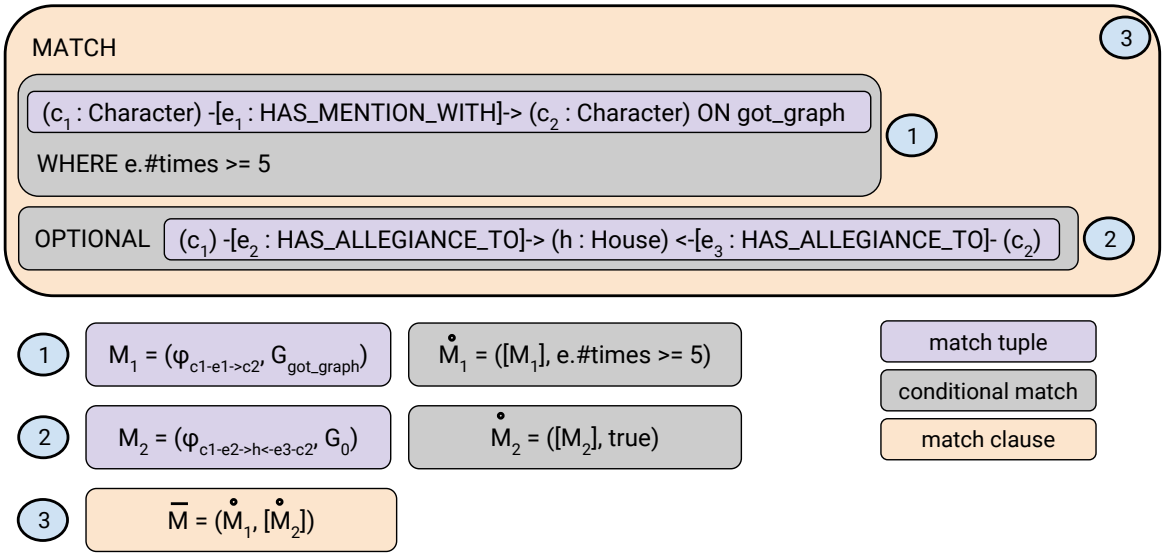

Figure 6.1: Breakdown of the algebraic representation of the `MATCH` block.

denomination syntax algebraic notation	notation breakdown
vertex pattern `(v)`, `(v:V)` $\pi_v = (v, L_v)$	The pattern of a vertex. v is the vertex reference, $L_v \subseteq L_N$ is a possibly empty set of labels.
edge pattern `-[e]->`, `-[e:E]->` $\pi_{a \xrightarrow{e} b} = (e, L_e, \pi_L, \pi_R, d)$	The pattern of an edge. e is the edge reference, $L_e \subseteq L_E$ is a possibly empty set of labels, π_L, π_R are the representations of the left and right endpoint, $d \in \{\rightarrow, \leftarrow\}$ is the direction.
path pattern `-/@p:P/->`, `-/p/->` $\pi_{a \xRightarrow{@_R} b} = (p, rt, L_p, q, \pi_L, \pi_R, d, o, c, r)$	The pattern of a path. p is the path reference; rt is a boolean, shows if the path is a reachability test; $L_p \subseteq L_P$ is a possibly empty set of labels; q is either the path quantifier `ALL` or $(n, distinct)$, with n the number of paths to compute and *distinct* a boolean; π_R, π_L are the representations of the left and right endpoint; $d \in \{\rightarrow, \leftarrow\}$ is the direction of the path; o is a boolean, shows whether the path is stored; c is either $\varnothing$ or a reference binding the cost of the path; r is either $\varnothing$ or a Kleene-star expression $\{l_r\}$ on the label l_r.
graph pattern $\pi_1\pi_2...\pi_n$ $\varphi_{\pi_1\pi_2..\pi_n} = [\pi_1, \pi_2, ...\pi_n]$	A (chained) sequence of one or more entity patterns. If $n = 1$, π_1 can be a vertex, edge or path pattern. If $n > 1$, π_i can only be an edge or a path pattern.
match tuple $\pi_1...\pi_k$ `ON` G_1, $\pi_{k+1}...\pi_{k+m}$ `ON` G_2, ... $M = (\varphi_{\pi_1\pi_2...\pi_n}, G)$	A graph pattern applied on a graph $G \in \{G_0, G_{\text{name}}\}$, where G_0 is the default graph and G_{name} is a graph specified through its name.
conditional match M `WHERE` ξ $\dot{M} = ([M_1, M_2, ...M_n], \xi)$	A list of match tuples conditioned by predicate ξ and with $n \geq 1$.
match clause `MATCH` $\dot{M}_1$ `OPTIONAL` $\dot{M}_2$ `OPTIONAL` $\dot{M}_2$ ... $\overline{M} = (\dot{M}_1, [\dot{M}_2, \dot{M}_3, ...\dot{M}_n])$	The entire `MATCH` block, with the non-optional conditional match and a possibly empty list of `OPTIONAL` conditional matches.

Table 6.2: `MATCH` algebra.

6.3.3 Label inference on graph patterns

We use the term *label inference* to refer to the process of finding the minimal set of labels for each variable in the `MATCH` block, given all the existing constraints of the block and the label constraints in the database $\mathcal{D}$. The G-CORE syntax allows variables to be specified without a label, however, because we store data per-label, we are interested in annotating each variable with at least one label, such that we identify which tables will be queried during the evaluation of the `MATCH` clause. It is essential to find the minimal set of labels, as the runtime of solving a query is directly related to the scan size of these tables.

Algorithm 1 presents a very basic approach we use in the interpreter for solving label inference. The input is a list containing all the match tuples in the `MATCH` block, i.e. the tuples in the non-optional conditional match, the tuples in the `OPTIONAL` sub-clauses and the tuples appearing in any `EXISTS` expression in the entire block. The algorithm splits each tuple into individual patterns (lines 2-4) and then creates an initial mapping ρ between each variable in each tuple and a set of labels (lines 8-15). The set of labels we choose is very loose, as we assign all the available vertex labels (L_N) to each vertex variable, all the available edge labels (L_E) to each edge variable, all the available stored path labels (L_P) to each stored path variable and all the available edge labels (L_E) to each virtual path[2] - as we only support simple Kleene-star expressions for the regular path expressions of virtual paths, all the edges along the path will have the same label in L_E.

A recursive function is then called (line 16) until the restrictions in ρ no longer change during the execution of its block. Within the function, we iterate over the list of patterns (line 19). If we encounter a vertex pattern we check whether the vertex had already been labeled in the pattern and, if it had, we update its mapping in ρ to the specific label, if necessary (lines 22-23). Otherwise, if we encounter an edge or a stored path pattern we try to update each pattern element iteratively (lines 24-35). First, we perform the same checks as for a simple vertex for the source endpoint (lines 25-29). This time, if there was no label specified in the pattern, (1) we extract the labels of the edge/path from ρ, (2) we extract the endpoint label tuple from each of these labels using the function τ[3], (3) we retain only the source label from each of these tuples into a set, (4) we compare this new set of labels to the vertex's mapping in ρ and update the mapping if it was different from the set we have just computed. After the source endpoint, we perform similar checks for the edge/path (lines 30-34), but this time, if the edge/path is not labeled in the pattern, we try to update its label by (1) considering all the labels in L_E or L_P (depending on the case), (2) extracting the source-destination label tuples with function τ for all these labels, (3) retaining only those tuples for which the source label is in ρ(source variable) and the destination label is in ρ(destination variable), (4) retaining the edge/path labels for which the source-destination label tuple has passed the previous test. After updating the edge/path, we try to update the mappings of the destination vertex exactly as we did for the source vertex. Finally, if we encounter a virtual path pattern, we do the same as for an edge or stored path, only that the virtual path label is already known, as is the edge label used in the Kleene-star expression.

Each time a mapping in ρ changes, a boolean value is set to `true`. At the end of the function, if this boolean was true, we call the function recursively. Otherwise, ρ is returned as the output of the algorithm, the restriction on all variables' labels we were aiming to compute. We provide a simple example of how the algorithm runs on a chained pattern in Figure 6.2.

[2]The label of a virtual path is symbolic - it is actually the label of the edges along the path.

[3]τ restricts an edge or stored path label to a fixed tuple of source and destination labels in our physical data model.

Algorithm 1: Label inference algorithm. We use $\leftarrow$ to denote that we assign the value on the right hand-side to the expression on the left hand-side. Tuple items are replaced with underscore when their value is not used in the algorithm. We express the algorithm with notations introduced in Table 6.2 and Section 4.3.

input : $[M_1, M_2, ...M_n]$ the list of all match tuples in the `MATCH` block
output: ρ a map from variable name to the minimal set of labels that determine it, for all variables in the `MATCH` block

begin
  $\Pi \leftarrow \varnothing$
  for $(\varphi_{\pi_1\pi_2...\pi_m}, G) \in [M_1, M_2, ...M_n]$ **do**
    for $\pi_k \in \pi_1\pi_2...\pi_m$ **do** $\Pi \leftarrow \Pi \cup (\pi_k, G)$
  $\rho \leftarrow$ `RestrictLabels(`Π`)`
  return ρ
Function `RestrictLabels(`Π`):`
  $\rho \leftarrow \varnothing$
  for $(\pi, G) \in \Pi$ **do**
    $(\_, \_, L_N, L_E, L_P, \_, \_) \leftarrow \mathcal{D}(G)$
    switch π **do**
      case vertex π_a **do** $\rho(a) \leftarrow L_N$
      case edge $\pi_{a \xrightarrow{e} b}$ **do** $\rho(a) \leftarrow L_N$, $\rho(b) \leftarrow L_N$, $\rho(e) \leftarrow L_E$
      case stored path $\pi_{a \xRightarrow{@_R} b}$ **do** $\rho(a) \leftarrow L_N$, $\rho(b) \leftarrow L_N$, $\rho(p) \leftarrow L_P$
      case virtual path $\pi_{a \xRightarrow{p} b}$ **do** $\rho(a) \leftarrow L_N$, $\rho(b) \leftarrow L_N$, $\rho(p) \leftarrow L_E$
  return `RestrictLabels` (Π, ρ)
Function `RestrictLabels(`Π, ρ`):`
  changed $\leftarrow$ `false`
  for $(\pi, G) \in \Pi$ **do**
    $(\_, \_, \_, \_, \_, \_, \tau) \leftarrow \mathcal{D}(G)$
    switch π **do**
      case vertex π_a **do**
        if $L_a = \{l_a\}$ *and* $\rho(a) \neq \{l_a\}$ **then** $\rho(a) \leftarrow \{l_a\}$, changed $\leftarrow$ `true`
      case edge or stored path $\pi_{a \xrightarrow{x} b}$ or $\pi_{a \xRightarrow{@_x} b}$ **do**
        if $L_a = \{l_a\}$ **then**
          if $\rho(a) \neq \{l_a\}$ **then** $\rho(a) \leftarrow \{l_a\}$, changed $\leftarrow$ `true`
        else // $L_a = \varnothing$
          $L'_a \leftarrow \{l_{\text{src}}, \forall (l_{\text{src}}, l_{\text{dst}}) \in \tau(\rho(x))\}$
          if $\rho(a) \neq L'_a$ **then** $\rho(a) \leftarrow L'_a$, changed $\leftarrow$ `true`
        if $L_x = \{l_x\}$ **then**
          if $\rho(x) \neq \{l_x\}$ **then** $\rho(x) \leftarrow \{l_x\}$, changed $\leftarrow$ `true`
        else // $L_x = \varnothing$
          $L'_x \leftarrow \{l_x, \forall l_x \in L_{E/P} : (l_{\text{src}}, l_{\text{dst}}) \leftarrow \tau(l_x) \vee l_{\text{src}} \in \rho(a) \vee l_{\text{dst}} \in \rho(b)\}$
          if $\rho(x) \neq L'_x$ **then** $\rho(x) \leftarrow L'_x$, changed $\leftarrow$ `true`
        update $\rho(b)$ similar to $\rho(a)$
      case virtual path $\pi_{a \xRightarrow{p} b}$ **do** // $r = \{l_r\}$
        update $\rho(a)$ as before
        if $\rho(p) \neq \{l_r\}$ **then** $\rho(p) \leftarrow \{l_r\}$, changed $\leftarrow$ `true`
        update $\rho(b)$ as before
  if *changed* **then return** `RestrictLabels(`Π, ρ`)` **else return** ρ

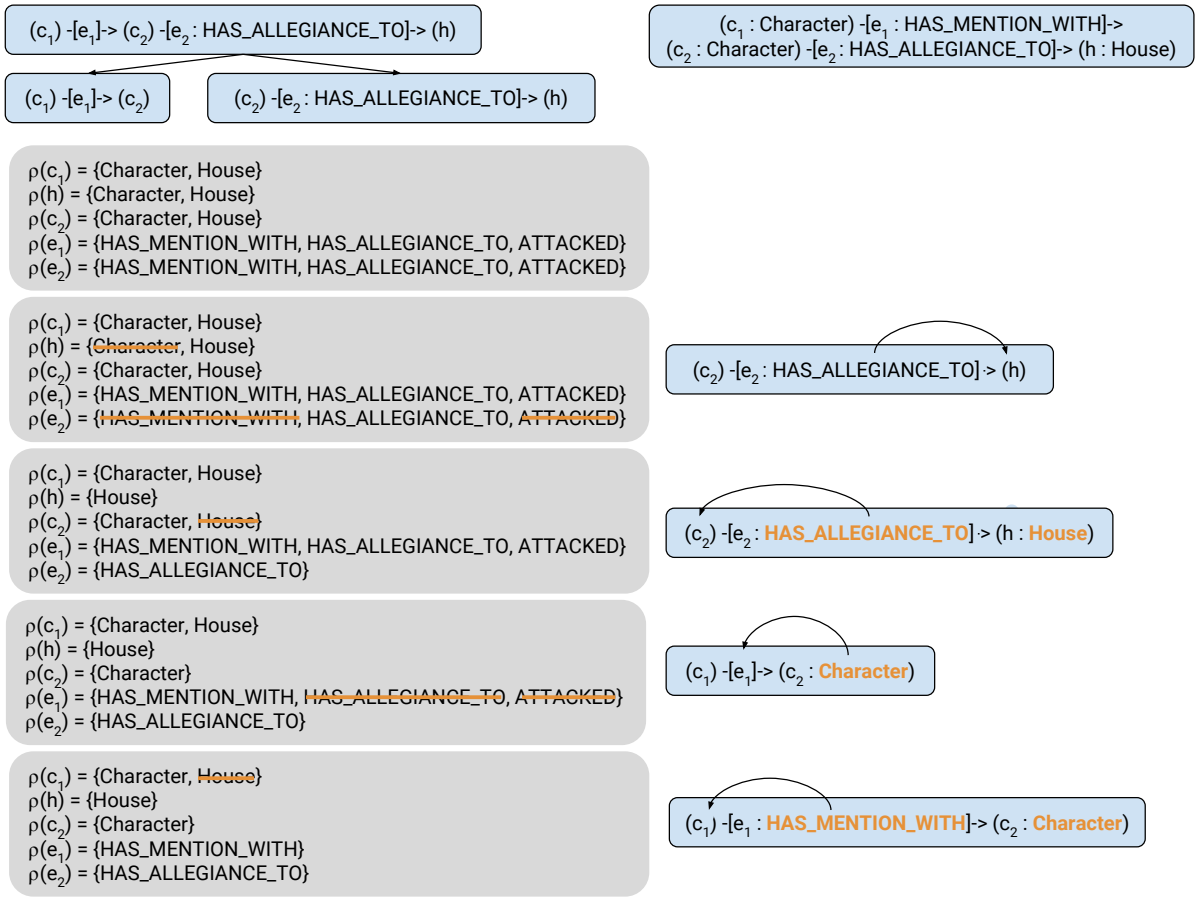

Figure 6.2: Label inference on a chained pattern. Initially, the restriction ρ for each variable is loose - all the available labels per item type. For each pattern tuple we try to update the restrictions of its variables. Source vertex is updated based on its strict label (if there is any) or on the edge restriction. Edge restriction is updated based on its strict label (if there is any) or on the restrictions of incident vertices. Destination vertex is updated based on its strict label (if there is any) or on the restriction of the edge. At each step, the restriction set for each variable either decreases in size, or remains constant. When no variable changes any more, the algorithm stops.

Complexity analysis

To determine the complexity of the entire process, we will look at worst-case scenarios for the function `RestrictLabels`. We start by determining the number of steps for one pattern to change the restrictions of its variables. For a vertex pattern, the vertex can be unlabeled, in which case the condition on line 23 is false and we return L_N. We assume this can be done in constant time $\mathcal{O}(1)$. If the vertex is strictly labeled in the pattern, the operation on line 23 can be computed in $\mathcal{O}(1)$ time. For an edge or path (stored or virtual), there are three checks/updates we perform. For the source vertex, if it is strictly labeled, we can solve it as before in $\mathcal{O}(1)$. Otherwise, we create a new set of labels on line 28 in $\mathcal{O}(max(|L_E|, |L_P|))$ ($\rho(x)$ has at most $|L_E|$ elements for an edge and at most $|L_P|$ elements for a path) and on line 29 we intersect this new set of labels with the restrictions of the source in $\mathcal{O}(|L_N| + max(|L_E|, |L_P|))$ steps (there are at most $|L_N|$ labels in the vertex's restriction). In total, there are $\mathcal{O}(|L_N| + 2 * max(|L_E|, |L_P|))$ we take to restrict the source's labels. The same applies for the destination vertex. For the edge or path variable, we will need $\mathcal{O}(1)$ if it was strictly labeled, or else we create a new label set on line 33 in

$\mathcal{O}(max(|L_E|, |L_P|))$ (because we iterate either L_E or L_P) and we intersect it with the previous restrictions on line 34 in $\mathcal{O}(max(|L_E|, |L_P|)+max(|L_E|, |L_P|)) = \mathcal{O}(2*max(|L_E|, |L_P|))$. In total, for a single edge or path, we will need $2*\mathcal{O}(|L_N|+2*max(|L_E|, |L_P|))+\mathcal{O}(2*max(|L_E|, |L_P|)) = \mathcal{O}(6*max(|L_E|, |L_P|)+2*|L_N|) = \mathcal{O}(max(|L_E|, |L_P|)+|L_N|)$ steps. This is also the worst-case complexity for any pattern type.

In the general case, we will restrict labels for more than one pattern in each match tuple. In this case, all the patterns are either paths or edges. Let $|Q_M|$ be the number of patterns across all match tuples in the `MATCH` block. At each step of the algorithm, the restrictions for each pattern either decrease, or remain constant. A change in the restrictions is either internal (caused by the restrictions of source/destination vertex or edge) or external (caused by the decrease in other patterns). In the beginning of the algorithm, each pattern can be labeled in $\mathcal{O}(max(|L_E|, |L_P|))$ ways, because we are also bound by the label restrictions in τ. We consider the worst possible case for restriction changes generated externally: in turn, each pattern decreases its number of labeling possibilities with one, triggered by a change in another pattern, until it converges to a single label combination. Internally, when a change occurs for any of its variables, a pattern will need the previous $\mathcal{O}(max(|L_E|, |L_P|)+|L_N|)$ steps to adjust the labels of its variables. Therefore, to update all $|Q_M|$ patterns we will need $\mathcal{O}(|Q_M|*max(|L_E|, |L_P|)*(max(|L_E|, |L_P|)+|L_N|)) = \mathcal{O}(|Q_M|*max(|L_E|, |L_P|)^2+|Q_M|*max(|L_E|, |L_P|)*|L_N|) = \mathcal{O}(|Q_M|*max(|L_N|, |L_E|, |L_P|)^2)$ steps.

6.3.4 Rewrite graph patterns

In the previous section we provided a basic algorithm for computing ρ, a mapping between each variable in the `MATCH` block and its minimal set of labels that satisfy the pattern constraints in the entire block, as well as the label constraints τ in the graph. Having acquired this information, we rewrite each match tuple M by splitting its graph pattern φ into individual, fully-labeled patterns. We wrap each such pattern into a match tuple. We provide the rewrite rules in Table 6.3.

As an outcome of these rewrite rules, the number of children of a conditional match increases (or, in the best case, it stays the same), however each of these children will be a match tuple with a single patterns in which each variable has been labeled. Note that multiple new match tuples may share the set of variables when their combination can be labeled in multiple ways. For example, the graph pattern `(a)-[e]->(b)` can be labeled in three ways in the got_graph: Character-HAS_MENTION_WITH-Character, Character-HAS_ALLEGIANCE_TO-House and House-ATTACKED-House.

Original operator	Rewrite rule
π_a	$(a, l_a), \forall l_a \in \rho(a)$
$\pi_{a \xrightarrow{e} b}$	$(e, l_e, (a, \tau(l_e).\_1), (b, \tau(l_e).\_2), \rightarrow), \forall l_e \in \rho(e)$
$\pi_{a \xRightarrow{@_P} b}$	$(p, rt, l_p, q, (a, \tau(l_p).\_1), (b, \tau(l_p).\_2), \rightarrow, \texttt{true}, c, \varnothing), \forall l_p \in \rho(p)$
$\pi_{a \xRightarrow{P} b}$	$(p, rt, \varnothing, q, (a, \tau(l_r).\_1), (b, \tau(l_r).\_2), \rightarrow, \texttt{false}, c, \{l_r\})$
$M = (\varphi_{\pi_1\pi_2...\pi_n}, G)$	$\{M_1, M_2, ...M_n\}$, where $M_k = (\varphi_{\pi_k}, G)$
$\dot{M} = ([M_1, M_2, ...M_n], \xi)$	$\dot{M} = ([M_1^1, M_1^2, ...M_1^{m_1}, M_2^1, M_2^2, ...M_2^{m_2}, ...M_n^1, M_n^2, ...M_n^{m_n}], \xi)$, where M_k has been rewritten to $\{M_k^1, M_k^2, ...M_k^{m_k}\}$

Table 6.3: Rewrite rules for graph patterns.

6.3.5 Rewrite conditional match

As we have seen in the previous sections, the children of the conditional match nodes have become match tuples of single vertex, edge or path patterns, in which all the variables are strictly labeled.

In this section we show how we rewrite the algebraic `MATCH` sub-tree from G-CORE operators to relational operators. During this rewrite phase we make no assumptions on the structure of the stored data tables, so the graph patterns are not changed and instead will be resolved at the target level. In a way, we can think of vertex, edge or path patterns as database scans, so we leave the actual scan operation to be fulfilled by a specific target, which is aware of the data layout.

The entire rewrite logic of this step will be based on the fact that match tuples of graph patterns of length one are the leaves of the relational tree. We view them conceptually as tables with data provided by the storage layer. The evaluation of each graph pattern is, in fact, a binding table on its own, therefore a relation over the variables in the pattern. To evaluate the entire `MATCH` block we then need to combine the binding tables of each graph pattern through relational operators.

We describe how conditional match operators are handled in Algorithm 2. Remember that the match tuples in the algorithm's input now contain graph patterns of only one graph item - vertex, edge or path. We introduce a function β on line 2 that, given an operator (match tuple or relational operator) yields the operator's binding set, i.e. a set that contains all the variable names that appear in that operator. For example, the pattern `(a)-[e]->(b)` has the binding set $\{a, e, b\}$. The binding set of a relational operator is the union of the binding sets of each graph pattern in that operator's sub-tree.

Starting from the list of match tuples in the input conditional match, we create a mapping B on line 4 between binding sets and the set of match tuples that share the binding set. As multiple combinations of labels can be applied to the same graph pattern, there can be cases when a binding set maps to more than one match tuple. Given that each match tuple is a binding table and that each of these binding tables will share the header, we can create a single unified binding table from all the match tuples that share the binding set by unioning their individual binding tables. On line 6 we define U, the set of all resulting union operators.

The initial list of match tuples has now become a list of union operators and the remaining match tuples that did not participate in any union. In other words, we are left with operators that have distinct binding sets. However, there can still be operators that share one or more variables between their binding sets. As each operator is a binding table, it means we can find tables that share one or more attributes. To combine them into a single binding table, we need to inner-join these operators on their common variable(s). For this, we first create V on lines 10-11, a mapping between a variable name and the set of operators that share the variable name in their binding set. Lines 12-20 change the mapping V in order to create the inner-joins. At each step, we take the first key v in V that is mapped to at least two operators (line 13). If such a key exists, we inner-join its values on line 15. After this, we need to update V for each variable name that appears in the join's binding set (line 16): we remove from its mapped operators the ones that are now part of the join (lines 17-18) and add the join operator itself to its mapping (line 19).

At the end of this computation, V will remain the same mapping between variable name and operators, however now each key in V will be mapped to exactly one operator - either (1) a match tuple that does not share any variables with any other match tuple in the input, (2) a union of match tuples that have the same binding set, but do not share any variable with any other match tuples in the input, or (3) the join of match tuples or unions that share at least one variable with at least one other operator in the join. If there is more than one unique operator left in V, we cross-join them to combine their respective binding tables.

Finally, the resulting operator (either match tuple, union, inner-join or cross-join) replaces the first member in the conditional match's tuple. We provide a simple example in Figure 6.3 showing how four graph patterns will be combined by Algorithm 2 into a sub-tree of relational operators. This is the sub-tree that becomes the first child of the rewritten conditional match.

Algorithm 2: Rewrites conditional match operators. The list of match tuples in a conditional match is replaced by a single relational operator, in which match tuples that share the binding set are unioned, operators that share at least one variable are inner-joined and operators that do not share any variable are cross-joined.

input : A conditional match operator $\dot{M} = ([M_1, M_2, ...M_n], \xi)$.
output: A modified $\dot{M}$ in which the list of match tuples has been replaced by the combination of the tuples under union, inner-join and cross-join operators.

begin
  $\beta(op)$ - the binding set of an operator (match tuple, union, join), i.e. a set containing all the variable names used by the operator
  B - a mapping between binding set and the set of match tuples that share the respective binding set. If b is a binding set, then $B(b) = \varnothing$ if $b \notin B$, i.e. for any key that is not in B, the result of accessing that key's value yields the empty set. Initially, B is the empty mapping $\varnothing$.
  for $M_k \in [M_1, M_2, ...M_n]$ **do** $b \leftarrow \beta(M_k)$, $B(b) \leftarrow B(b) \cup \{M_k\}$
  U - the set of match tuples that will be combined under the union operator
  $U \leftarrow \bigcup_{M_k \in B(b)} M_k, \forall b \in B$ for which $B(b)$ has size ≥ 2
  J - the set of operators (match tuples or unions) that will be combined under the inner-join operator
  M_u - the initial list of match tuples in $\dot{M}$ minus all the match tuples used in U
  V - a mapping between variable name and the set of operators (match tuple, union or join) that have the respective variable in their binding set. V has the same properties as B.
  for $op \in M_u \cup U$ **do**
    for $v \in \beta(op)$ **do** $V(v) \leftarrow V(v) \cup \{op\}$
  repeat
    $v \leftarrow$ first variable name in V for which $V(v)$ has size ≥ 2, or $\varnothing$ otherwise
    if $v \neq \varnothing$ **then**
      $ops_j \leftarrow V(v)$, $j \leftarrow \bowtie_{op_j \in ops_j} op_j$, $J \leftarrow J \cup \{j\}$
      for $w \in \beta(j)$ **do**
        $ops_i \leftarrow ops_j \cap V(w)$
        for $op_i \in ops_i$ **do** $V(w) \leftarrow V(w) \setminus \{op_i\}$
        $V(w) \leftarrow V(w) \cup \{j\}$
  until $v \neq \varnothing$
  C - the final result, the combination of all operators left in V (match tuples, unions or joins) under the cross-join operator, if there is more than one operator
  V - the previous mapping that has been updated with joined operators. Note that after computing J, every variable name in V will be mapped to exactly one operator. Also note that each variable that appears in one graph pattern will be mapped to the same operator, therefore we use set union below to discard duplicates from the final result.
  $C \leftarrow \times_{op \in ops} op$, where ops are all the unique operators in V, if ops has the size ≥ 2
  rewrite $\dot{M}$ to the tuple (C, ξ)
  return the new $\dot{M}$

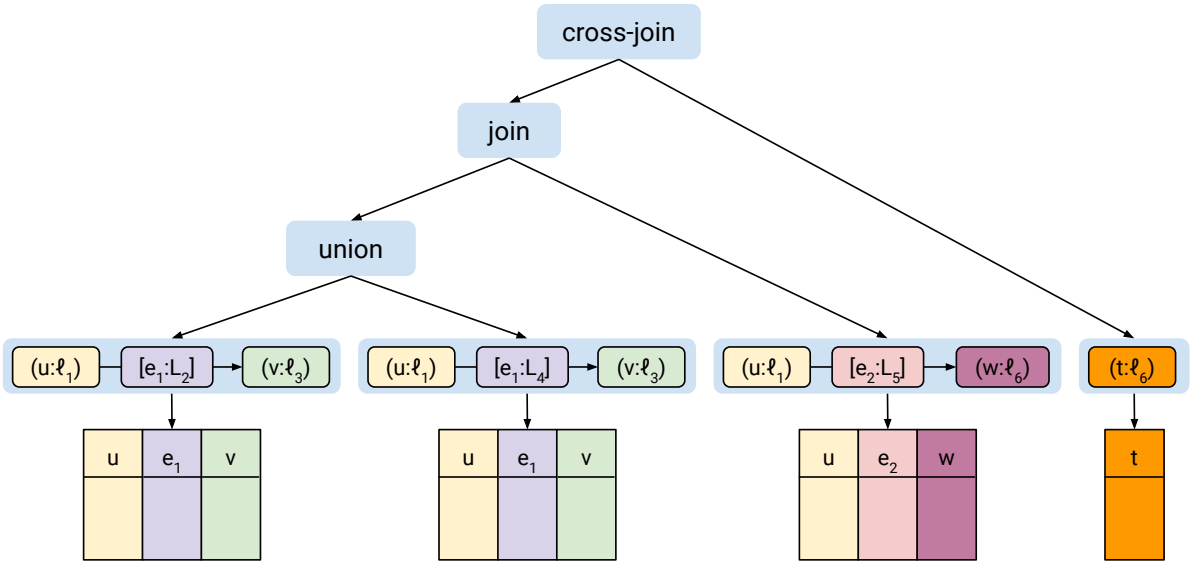

Figure 6.3: Example of how a conditional match operator is rewritten. Its list of match tuples becomes one relational operator, in which the tuples that have the same binding set are unioned, operators that share at least one variable are inner-joined and operators that share no variable are cross-joined. The result is a left-deep algebraic plan.

Complexity analysis

To estimate the number of steps needed to reach convergence, we will again look at worst-case scenarios. Let $|Q_M|$ be the number of match tuples in the input. Building B on line 4 costs $\mathcal{O}(|Q_M|)$ and building U on line 6 costs $\mathcal{O}(|Q_M|^2)$ - we consider the cost of building a left-deep union sub-tree as $\mathcal{O}(|Q_M|)$. The cost of building V on lines 10-11 is $\mathcal{O}(3 * |Q_M|)$, as each match tuple can have at most three variables (edge or path), all distinct from other match tuples. To create J we will need $\mathcal{O}(|Q_M|^3)$ steps - we find v on line 13 in $\mathcal{O}(|Q_M|)$; we iterate over each variable name in j on line 16 in $\mathcal{O}(3 * |Q_M|)$ and need $\mathcal{O}(|Q_M|)$ steps for each such variable on line 18 to update its set of operators; in the worst case, the condition on line 13 passes at each iteration for all $3 * |Q_M|$ variables. In total, we will need $\mathcal{O}(|Q_M|^3)$ steps to merge the n match tuples of a conditional match.

6.3.6 Rewrite MATCH operators

Using the previous rewrite phase, we can now rewrite each conditional match into a tuple consisting of a sub-tree of relational operators that combine the match tuples of the conditional match, and an expressions that should act as a predicate over the evaluation of the relational sub-tree. We now have all the ingredients to rewrite the entire `MATCH` block. We do this bottom-up with the rewrite rules summarized in Table 6.4.

A conditional match over a relation becomes a selection over that relation, because the semantics of the `WHERE` sub-clause are that the expression in the sub-clause is used to filter the bindings generated by its attached list of match tuples. This rewrite applies over all conditional matches in the `MATCH` block, therefore on both non-optional and optional sub-clauses. The entire `MATCH` block then becomes the left-outer-join of all the block's conditional matches, starting from the non-optional clause.

Original operator	Rewrite rule
$\dot{M} = ([M_1, M_2, ...M_n], \xi)$	$\dot{M} = (C, \xi)$, where C is the operator returned by Algorithm 2
$\dot{M} = (C, \xi)$	$\sigma_\xi(C)$
$\dot{M} = (\dot{M_1}, [\dot{M_2}, \dot{M_3}, ...\dot{M_n}])$	$\bowtie_{M_k \in [\dot{M_1}, \dot{M_2}, \dot{M_3}, ...\dot{M_n}]} M_k$

Table 6.4: Rewrite rules of MATCH operators.

6.4 The CONSTRUCT clause

The purpose of the CONSTRUCT clause is to provide rules on how to build a new graph from the bindings produced in the MATCH clause. These rules act as a "stamping pattern" on the binding table, from which new PPGs are constructed: for each binding row it generates graph content by "stamping" the pattern with bound parameters into the resulting graph. The CONSTRUCT clause contains a non-empty list of graphs: named graphs and/or sub-queries and/or graph construct patterns. The list of graphs is followed by zero or more SET clauses that add properties or labels to the construct variables, and zero or more REMOVE clauses that remove properties or labels from construct variables. The evaluation of the entire CONSTRUCT block must produce one PPG, which is the graph union of all the graphs provided as arguments.

As we previously did with the MATCH clause, we initially create the CONSTRUCT sub-tree from G-CORE algebraic operators that closely follow G-CORE's grammar defined at [7] and subsequently reshape this tree into relational operators.

6.4.1 Algebraic representation of vertices, edges and paths

Vertex, edge and path patterns specified in the CONSTRUCT block have a different meaning from their MATCH counterpart - they act as rules for how to create the new entities in the constructed graph. However, from an implementation point of view, they are still nodes in the algebraic tree and we can again model them as tuples of various items. We will use κ to denote the construct pattern of an entity.

The construct pattern of a vertex is the tuple $(v, \gamma, \lambda_S, \sigma_S)$, where v represents the variable name used in the pattern, γ is the group declaration of the vertex or the empty set, if no grouping was explicitly given, and λ_S and σ_S represent the set of inline label SET assignments and the set of inline property SET assignments[4], respectively, or else the empty set if there are no inline assignments for this vertex. Construct patterns can also specify a copy pattern for the new entities, but, as shown in Table 4.1 we only parse copy patterns and do not yet implement them, so for simplicity we omit them from the construct tuple. The group declaration is represented through a set that can contain property references $x.k$, where x must be a variable bound in MATCH and k is a property key, or simple variable references. Explicit groupings provided under the GROUP key-word in a pattern can only be property references, but we will see in later rewrite phases that we also need to accept variable names in this set. We will denote the assignment of label l to the vertex v with $(+v : l)$ and the assignment of the property-key k to value ξ for the vertex v with $(+v.k = \xi)$, where ξ is an expression.

For example, the vertex construct pattern `(c)` is the tuple $\kappa_c = (c, \varnothing, \varnothing, \varnothing)$, while the pattern `(b GROUP a.battle_name :Battle {name := a.battle_name})` is the construct tuple $\kappa_b = (b, \{a.\text{battle\_name}\}, \{(+b : \text{Battle})\}, \{(+b.\text{battle\_name} = a.\text{battle\_name})\})$.

The construct pattern of an edge is the tuple $(e, \kappa_L, \kappa_R, d, \gamma, \lambda_S, \sigma_S)$, where e is the variable name used in the pattern, κ_L and κ_R and the construct patterns of the right and left endpoint,

[4]Recall that in the PPG model described in Section 4.2 λ is the function that maps an entity identifier to its set of labels, while σ is the function that maps an entity identifier and a property key to a value.

$d \in \{\rightarrow, \leftarrow\}$ is the edge's direction, and γ, λ_S and σ_S hold the same meaning as before. As in the MATCH clause, we do not support undirected or bidirectional edges, even though they are syntactically valid in G-CORE queries. For example, we would represent the edge construct pattern `(a)-[e]->(b)` as $\kappa_{a \rightarrow eb} = (e, (a, \varnothing, \varnothing, \varnothing), (b, \varnothing, \varnothing, \varnothing), \rightarrow, \varnothing, \varnothing, \varnothing)$ and the pattern `(b)<-[w :WAS_IN {role := "attacker"}]-(h)` as the tuple $\kappa_{h \rightarrow wb} = (w, (b, \varnothing, \varnothing, \varnothing), (h, \varnothing, \varnothing, \varnothing), \leftarrow, \varnothing, \{(+w : \text{WAS\_IN})\}, \{(+w.\text{role} = \text{"attacker"})\})$.

While we did not yet implement the construction of paths, we still parse their construct patterns and add these to the algebraic tree. We represent the patterns of paths as the tuple $(p, \kappa_L, \kappa_R, d, o, \gamma, \lambda_S, \sigma_S)$, where p is the variable used in the pattern, $\kappa_L, \kappa_R, d, \gamma, \lambda_S, \sigma_S$ hold the same meaning as for an edge - however, γ in this case must be empty in the parse tree (path patterns do not allow the GROUP keyword), but leave it as part of the tuple because we can populate it with an implicit value during the rewrite phases. o is a boolean value which tells us whether the path is stored (`true`) or virtual (`false`). For virtual paths, λ_S and σ_S must be the empty set. As an example, the construct pattern `(c)-/@p :CATELYN_TO_DROGO {#hops := cst}/->(d)` will become the tuple $\kappa_{c \Rightarrow @pd} = (p, (c, \varnothing, \varnothing, \varnothing), (d, \varnothing, \varnothing, \varnothing), \rightarrow, \texttt{true}, \{(+p : \text{CATELYN\_TO\_DROGO})\}, \{(+p.\#\text{hops} = \text{cst})\})$.

Each entity construct pattern will always have the variable name defined - it was either specified in the query, or otherwise replaced with a fresh name generated by the canonical rewrite of the parse tree.

6.4.2 CONSTRUCT operators

The parsed CONSTRUCT block is translated into a tree-like representation of G-CORE operators, which becomes the CONSTRUCT sub-tree in the algebraic tree. Exactly as we broke down the MATCH block into several operators, we will split CONSTRUCT into self-contained sub-clauses.

We start by defining the *graph construct pattern* as either a vertex construct pattern, or a vertex construct pattern followed by zero or more connection construct patterns. A connection construct pattern is either the construct pattern of an edge, or that of a path, followed by the construct pattern of the right endpoint. Therefore, graph construct patterns can define graph shapes of various complexities, from simple vertices to intricate edge or path topologies. We will use $\psi_{\kappa_1 \kappa_2 ... \kappa_n} = [\kappa_1, \kappa_2, ...\kappa_n]$ to denote the algebraic representation of a construct pattern consisting of $n \geq 1$ individual entity patterns, linked into a graph topology. Its algebraic form is an n-arry list: if $n = 1$, then the single element κ_1 can only be a vertex construct pattern, otherwise, if $n > 1$, the elements of the list can only be edge or path construct patterns. Graph construct patterns are similar to the graph patterns φ we introduced for the MATCH clause.

We now introduce the first G-CORE operator used in the CONSTRUCT clause, namely the *conditional construct* $\dot{C} = (\psi_{\kappa_1 \kappa_2 ... \kappa_n}, \xi)$. It represents the most basic sub-clause in CONSTRUCT, which specifies a graph construct pattern which has to be applied on the binding table filtered by the condition ξ. Yet again, ξ is an expression. When the condition is not specified in the query, we will fill it in as `true` for uniformity. For example, the construction `(a)-[e]->(b)` becomes $\dot{C} = (\psi_{a \xrightarrow{e} b}, \texttt{true})$, while `(a)-[e]->(b) WHEN a.prop >= 2` is $\dot{C} = (\psi_{a \xrightarrow{e} b}, \texttt{a.prop >= 2})$.

The second and final G-CORE operator we introduce is the *construct clause* $\overline{C} = ([\dot{C}_1, \dot{C}_2, ... \dot{C}_n], \lambda_S, \lambda_R, \sigma_S, \sigma_R)$, comprised of $n \geq 1$ conditional constructs, label and property SET assignments λ_S and σ_S and label and property REMOVE assignments λ_R and σ_R. When previously defining entity construct patterns we introduces the notion of inline label and property assignment, which can only add labels or properties to the variable they are inlined with. Labels and properties can also be added to any variable that appears in the CONSTRUCT block with the help of SET sub-clauses and we will use the same notation as before for them: $(+x : l)$ denotes that variable x is added a new label l and $(+x.k = \xi)$ denotes that variable x is assigned a new property with key k and value ξ. The REMOVE sub-clauses define the labels or properties to be

removed from construct variables and we will use the $(-x : l)$ and $(-x.k)$ notations for these REMOVE assignments, with the obvious meaning. Note that the canonical `CONSTRUCT` clause in G-CORE, as designed in [20], also takes graph names or graph queries as arguments. As we did not implement this feature, we left its representation aside from the operator we have just defined.

Figure 6.4 presents the breakdown of the `CONSTRUCT` clause into its specific operators and Table 6.5 summarizes the concepts introduced in this section.

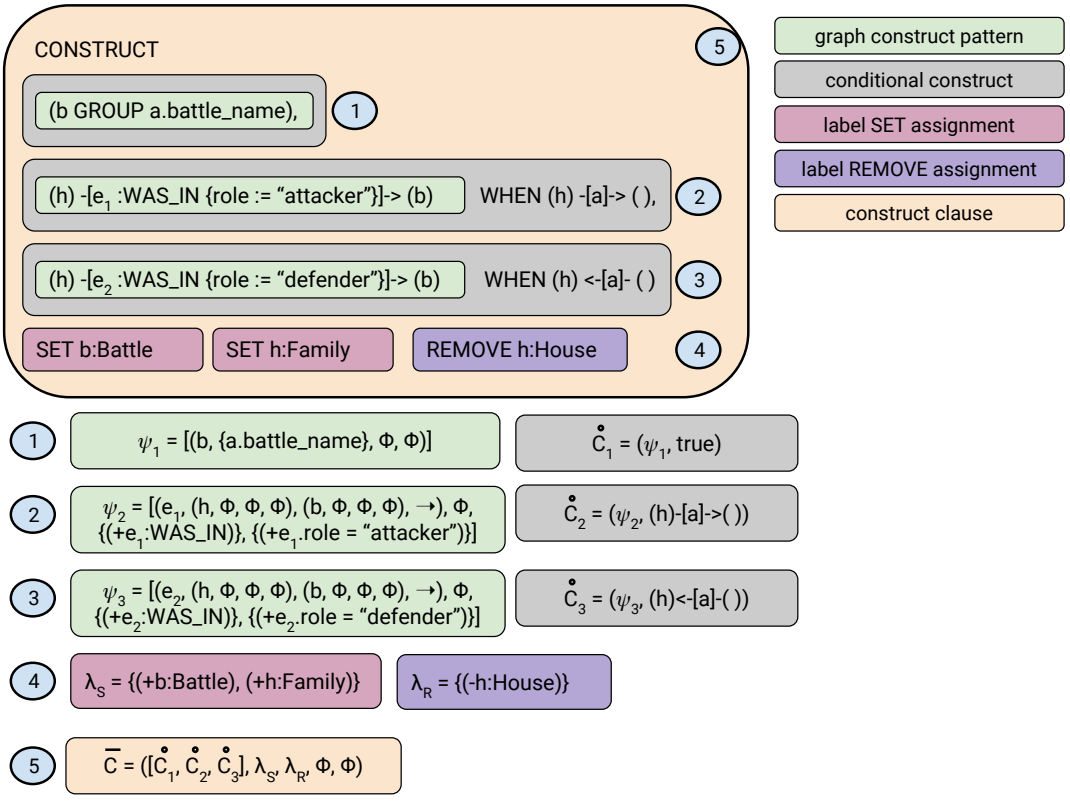

Figure 6.4: Breakdown of the algebraic representation of the `CONSTRUCT` block.

6.4.3 Vertex and edge creation

Graph construct patterns contain multiple vertex construct patterns, interlinked through edge or path construct patterns. As we did not implement path creation, we will not discuss it in this section. We will focus on how new vertices and edges can be created, by starting with the simplest case - the vertex construct pattern. Let v be the construct variable of the vertex. The construct pattern is evaluated in the presence of the binding table created by the `MATCH` clause. We denote the binding table with Ω and, as stated before, we view it as an unnamed relation $\Omega = (H, B)$, with a header H and a body B. The header contains the set of all variables that participate in the `MATCH` block and the body contains the bindings of each of the variables. We distinguish three cases for vertex construction, which we discuss in turn below.

Case 1. We are creating a vertex using a variable name that has already been matched in the binding table, so $v \in H$ and there are bindings of v in B. The identity of the vertex is given by its unique identifier in $\mathcal{D}(G)$, so we consider it an error for the query to specify an explicit `GROUP`-ing for bound variables, because vertices with different identifiers but equal grouping attributes would be coalesced into a single entity through this aggregation, effectively violating the identities of the participating bindings. However, in this case, there is an implicit

denomination syntax algebraic notation	notation breakdown
label SET assignment `(x:l)`, `SET x:l` $(+x:l)$	Construct variable x is assigned the label l, which is added to its set of labels.
property SET assignment `(x {k := v})`, `SET x.k := v` $(+x.k = v)$	Construct variable x is assigned a new property, with key k and value v.
label REMOVE assignment `REMOVE x:l` $(-x:l)$	Label l is removed from construct variable x's set of labels.
property REMOVE assignment `REMOVE x.k` $(-x.k)$	Property k is removed from the construct variable x's set of properties.
vertex construct pattern `(v)`, `(v :l {k := expr})` $\kappa_v = (v, \gamma, \lambda_S, \sigma_S)$	The construct pattern of a vertex. v is the vertex reference, γ is a possibly empty set of explicit grouping attributes, λ_S is a possibly empty set of inline label assignments and σ_S is a possibly empty set of inline property assignments. The elements of γ are property references $x.k$, where x is one of the matched variables and k a property key of x.
edge construct pattern `-[e]->`, `-[e :l {k := expr}]->` $\kappa_{a \xrightarrow{e} b} = (e, \kappa_L, \kappa_R, d, \gamma, \lambda_S, \sigma_S)$	The construct pattern of an edge. e is the edge reference, κ_L and κ_R are the representations of the left and right endpoint, $d \in \{\leftarrow, \rightarrow\}$ is the direction of the edge and γ, λ_S and σ_S hold the same meaning as for a vertex construct pattern.
path construct pattern `-/p/->`, `-/@p :l {k := expr}/->` $\kappa_{a \xRightarrow{@_p} b} = (p, \kappa_L, \kappa_R, d, o, \gamma, \lambda_S, \sigma_S)$	The construct pattern of a path. p is the path reference, κ_L and κ_R are the representations of the left and right endpoint, $d \in \{\leftarrow, \rightarrow\}$ is the direction of the path, o is a boolean that shows whether the path is stored and γ, λ_S and σ_S hold the same meaning as for the vertex and construct patterns. For paths, γ must be $\varnothing$ in the parse tree. For virtual paths ($o =$ `false`) λ_S and σ_S must be empty as well.
graph construct pattern $\kappa_1 \kappa_2 ... \kappa_n$ $\psi_{\kappa_1 \kappa_2 ... \kappa_n} = [\kappa_1, \kappa_2, ... \kappa_n]$	The entire chain of individual construct patterns that define the shape of a new graph. If $n = 1$, κ_1 can be a vertex, edge or path construct pattern. If $n > 1$, κ_i can only be an edge or path construct pattern.
conditional construct $\kappa_1 \kappa_2 ... \kappa_n$ `WHEN` ξ $\dot{C} = (\psi_{\kappa_1 \kappa_2 ... \kappa_n}, \xi)$	A graph construct pattern built from the binding table on which condition ξ is applied.
construct clause `CONSTRUCT` $\dot{C}_1, \dot{C}_2, ... \dot{C}_n$ `SET` λ_S / σ_S `REMOVE` λ_R / σ_R $\overline{C} = ([\dot{C}_1, \dot{C}_2, ... \dot{C}_n], \lambda_S, \lambda_R, \sigma_S, \sigma_R)$	The entire `CONSTRUCT` block, with n conditional constructs and the `SET` and `REMOVE` sub-clauses.

Table 6.5: `CONSTRUCT` algebra.

grouping of the binding using v's identity. Grouping by v is essential because the binding table might contain duplicate values bindings v. These duplicates can occur when, for example, non-correlated match tuples are used in `MATCH`, which leads to a cross-join of the respective bindings, or when edges originating from the same endpoint v are matched. The new vertex must only be created from the unique bindings of v, hence the necessary aggregation. Given the abstract view of the binding table we use in the algebraic module, this simply means that we need to group the binding table by v itself and the creation of a new vertex in this case starts from the relation $\Gamma_v(\Omega)$.

Case 2. We are creating a vertex using a variable name that has not been matched in the binding table and does not have explicit `GROUP`-ing, so $v \notin H$. Moreover, γ, v's set of explicit grouping attributes, is empty. No aggregation is necessary in this case and instead we use v's construct pattern as a "stamping" device on the binding table, in that for each individual binding tuple we create a new vertex. Therefore, the creation of a new unbound vertex starts from the binding table Ω to which we add a new column v, in which the values are new vertices with unique identities. For simplicity, we will denote this operation with $\Pi_{H \cup \{v\}}(\Omega)$, a projection on the binding table of all the attributes in its header H, plus the new variable v. Note that because there is no grouping in this case, using aggregate expressions to set new properties for v is considered an error.

Case 3. We are creating a vertex using a variable name that has not been matched in the binding table and has explicit `GROUP`-ing. As in the previous case, $v \notin H$, but now $\gamma \neq \varnothing$. In this case, the creation of the new vertex will start from the relation $\Pi_{H \cup \{v\}}(\Gamma_{\gamma; f_1(a_1), \ldots fn(a_n)}(\Omega))$, in which we group the binding table by the grouping attributes specified in γ and subsequently add to the result a new column with fresh bindings for v. This is also a case in which properties can be added to the new vertices by way of aggregate expressions $f_1, f_2, \ldots f_n$ on property references of bound variables $a_1, a_2, \ldots a_n \notin \gamma$.

Once we established the creation rules for vertices, we can define them for edges as well. The creation of an edge starts with the creation of its two incident vertices. Let $\kappa_{a \xrightarrow{e} b}$ be the construct pattern of the edge we are creating. The binding table is implicitly grouped by the identities of the the two endpoints a and b, regardless of whether the edge has been matched or not. If we are creating the edge $e \in H$, a and b must also be in H, otherwise the edge's identity is violated - if a and b would not be bound variables, the same matched vertex e would become a link between two different pairs of vertices. To create the edge, we must first create both a and b, such that their identities are available for grouping. If $e \in H$, we simply keep e's existing identities. Otherwise, we add a new column to the grouped binding table with fresh identities for e. Additionally, edge construct patterns for edges $e \notin H$ can include an explicit `GROUP` declaration. In this case, the grouping attributes are added to the aggregation key established before.

6.4.4 Rewrite conditional construct

We saw that the vertex and edge construct patterns use implicit grouping of the matched variables to create new unique entities. Additionally, explicit `GROUP`ing can be used by hand for custom aggregation of the binding table. With this in mind, the definition of the `WHEN` subclause in a conditional construct becomes somewhat ambiguous in [20]. On the one hand, it can accomplish a similar role to that of `WHERE` and filter the binding table before creating the new graph elements. On the other hand, it can also serve the role of the SQL HAVING clause, which filters the result of an aggregation. Indeed, examples of both interpretations are provided in [20]. In our implementation, we decide to treat `WHEN` exactly as the `WHERE` clause, so it becomes a pre-aggregation filter operation.

Let $\dot{C} = (\psi_{\kappa_1\kappa_2\ldots\kappa_n}, \xi)$ be a conditional construct we translate into relational operators. Algorithm 3 shows how $\dot{C}$ is resolved starting from the binding table Ω and the label and property SET and REMOVE assignments of the outer construct clause. It is important to note that at this stage in the rewriting process we do not have access to the actual data of the binding table - this data will only be computed in the target module, when the rewritten `MATCH` block will be evaluated against the graph database. We solve this issue by introducing a TableView node to act as a proxy for the binding table and apply all the relational operations on this view. The target is responsible for registering the view on the materialized binding table before starting to execute the construct queries.

The algorithm starts on line 2 by creating a mapping between construct variable and all its inline label and property SET and REMOVE assignments and from the assignments of the outer construct clause that pertain to that variable. For example, in the query `CONSTRUCT (a { prop1 := b.prop1 + 1 }) SET a.prop2 := b.prop2 * 3`, a's property SET assignments are both $(+a.\text{prop1} = b.\text{prop1} + 1)$ and $(+a.\text{prop2} = b.\text{prop2} * 3)$.

We first create all the vertices in the conditional construct (lines 7-13). Ω_V is the table that will store all the vertices and we build it in the same time as we generate the vertex create rules. The process starts from a selection over the binding table by expression ξ (line 6). Here we emphasize again that ξ is an expression that filters the binding table before starting the variable creation, so aggregates are not allowed in the expression sub-tree. On line 9 we treat vertex variables that do not appear in the binding table's header and do not have explicit aggregation (case 2 from the previous section) and add one new column to Ω_V for each of them. For vertex variables that appear in the binding table's header (case 1 from the previous section) we group the filtered binding table by the vertex identity on line 11. For vertex variables that do not appear in the binding table's header but have explicit grouping (case 3 from the previous section) we first group the filtered binding table by the explicit grouping attributes, then add a new column to the result with fresh bindings. For cases 2 and 3 we use the previous convention of expressing the column addition by projecting the new variable from the binding table, even though that variable does not appear in the header of the table. We will respect this convention when translating to SQL queries in the target module.

On line 11, when creating the new relation for case 3, we also pass to the group by operation property SET assignments that use aggregate expressions. In our implementation we pass the entire expression sub-tree, if it contains an aggregation at any of its levels. As we eventually translate the algebraic operators into SQL queries, we will later on make use of SQL's feature of computing column aggregates during the group by process.

Each new vertex relation built for case 3 is joined back to Ω_V on line 12, to bring the vertex's fresh bindings into the table. It is necessary to use the grouping attributes γ as the join condition. These are property references of the bound variables, therefore they will exist in the binding table. Hence, on line 11, the right argument of the join is a projection of the grouping attribute set γ and the new vertex from the vertex's construct relation. This is a case where we need to break away from the abstracted view of the binding table presented in Section 4.5, but it is nonetheless necessary: when aggregating the binding table, properties that are not part of the grouping key will be coalesced into a single value for equal keys and will lose their original meaning. Therefore, if we used them in the join key afterwards, the join would not make sense. It is only the aggregation keys that are safe to use for this purpose and we pass this information to the target by adding γ to the projection set.

The set of construct rules for vertex variables is expanded on line 13 with the relation from which the vertex can be created and its information from $\sigma_S, \sigma_R, \lambda_S, \lambda_R$. It is the target's task to make to use the construct relation and to create the vertex, i.e. to add or remove labels and properties from the vertex. The construct relation we create still contains the bindings of the matched variables, so their properties can be used to generate new ones for the vertex.

Algorithm 3: Rewrites a conditional construct operator to a tuple of two sets of construct rules, one for the vertex and one for the edge construct variables in the clause. A construct rule is a tuple that contains the construct variable, a relation from which the new entity can be built, along the label and property SET and REMOVE assignments for that entity - both inline assignments and the ones in the `SET` and `REMOVE` sub-clauses.

input : A conditional construct operator $\dot{C} = (\psi_{\kappa_1\kappa_2\ldots\kappa_n}, \xi)$, the binding table $\Omega = (H, B)$, the label and property SET and REMOVE assignments $\lambda_S, \lambda_R, \sigma_S, \sigma_R$, from `SET` and `REMOVE` sub-clauses.

output: A modified $\dot{C}$ that becomes a tuple of two sets with construct rules, one for the vertex construct variables in $\dot{C}$ and one for the edge construct variables in $\dot{C}$. Each construct rule is a tuple containing the construct variable, a relation from which one vertex or edge can be built, as well as its label and property SET and REMOVE assignments.

begin

- $\lambda_S, \lambda_R, \sigma_S, \sigma_R$ are transformed such that now they map variable constructs to their respective label and property SET and REMOVE assignment both in inline patterns and in the initial $\lambda_S, \lambda_R, \sigma_S, \sigma_R$. For example, $\lambda_S(v)$ will yield all of v's inline label SET assignments in $\dot{C}$ and all of v's label SET assignments in the `SET` sub-clause.
- V - the set of vertex construct variables in $\dot{C}$
- E - the set of edge construct variables in $\dot{C}$
- V_C, E_C - the sets of construct rules for the vertex and edge construct variables in $\dot{C}$. Initially, $V_C, E_C \leftarrow \varnothing$.
- $\Omega' \leftarrow \sigma_\xi(\Omega)$, the binding table filtered by condition ξ
- $V_{\text{no\_grp}} \leftarrow \{v \in V : v \notin H \text{ and } \gamma = \varnothing\}$, where γ is v's grouping attributes set
- $V_{\text{grp}} \leftarrow V \setminus V_{\text{no\_grp}}$
- $\Omega_V \leftarrow \Pi_{H \cup V_{\text{no\_grp}}}(\Omega')$, the binding table filtered by condition ξ to which all new vertices have been added
- **for** $v \in V_{grp}$ **do**
  - $r \leftarrow$ **if** $v \in H$ **then** $\Gamma_v(\Omega')$ **else** $\Pi_{H \cup \{v\}}(\Gamma_{\gamma; f_1(a_1), \ldots f_n(a_n)}(\Omega'))$
  - **if** $v \in H$ **then** $\Omega_V \leftarrow \Omega_V \bowtie_\gamma \Pi_{\gamma \cup v}(r)$
  - $V_C \leftarrow V_C \cup \{(v, r, \lambda_S(v), \lambda_R(v), \sigma_S(v), \sigma_R(v))\}$
- **for** $e \in E$ **do** // a, b endpoints of e
  - $r \leftarrow$ **if** $e \in H$ **then** $\Gamma_{a,b,e}(\Omega_V)$ **else** $\Pi_{H \cup \{e\}}(\Gamma_{a,b}(\Omega_V))$
  - $E_C \leftarrow E_C \cup \{(a, b, e, r, \lambda_S(e), \lambda_R(e), \sigma_S(e), \sigma_R(e))\}$
- rewrite $\dot{C}$ to tuple (V_C, E_C)
- **return** the new $\dot{C}$

Once all the vertices in the conditional construct appear as bindings in Ω_V, we can create the edges (lines 14-16). If the edge has already been used in the `MATCH` clause, we group the construct table and use the identity of the edge and of as the group by key. As multiple edges can occur between a pair of endpoints, we need to add the edge as well to the key. Otherwise, if the edge had not been bound, we group by the identities of the two endpoints and add a new column to the result with fresh bindings for the edge. The construct rule for the new edge is then added to the list of edge rules.

The final step in the rewrite phase is to rewrite the conditional construct operator to a tuple of construct rules for vertices and edges on line 17. We will now analyze the number of operations applied to the *filtered* binding table to create a variable. For vertices, we will need one operation if the vertex was unmatched and did not have explicit aggregation (a column addition); one operation if the vertex was matched (group by); three operations if the vertex was unmatched and had explicit aggregation (group by, followed by a column addition and a join). For edges, we will need one operation if the edge was matched (group by) or two operations otherwise (group by followed by column addition). The target will need to add its own projections to extract the vertex and edge information from its construct relation.

Complexity analysis

Once again, we examine the number of steps needed by the previous algorithm to rewrite a conditional construct operator. The rewrite starts on line 2 with the creation of the mapping between variable and its respective SET and REMOVE assignments in the entire `CONSTRUCT` block. We need $\mathcal{O}(|Q_C|)$ steps to achieve this, where $|Q_C|$ represents the number of construct variables in the conditional construct. Lines 7, 8 and 9 require an iteration over the set of vertex construct variables in the conditional construct, or over a subset of this, therefore, each will require $\mathcal{O}(|Q_C|)$ time. Each of the two for-loops on lines 10-13 and 14-16 will evaluate in $\mathcal{O}(|Q_C|)$ steps - we make the assumption that testing whether a variable is present in the header of the binding table can be achieved in constant time. Overall this means that the runtime complexity of Algorithm 3 is $\mathcal{O}(|Q_C|)$.

6.4.5 Rewrite CONSTRUCT operators

The rewriting of the `CONSTRUCT` block is done bottom-up by first rewriting each conditional construct in the block as described in the previous section. The enclosing construct clause is then transformed with Algorithm 4, in which all the vertex construct rules and all the edge construct rules from all the conditional construct operators are gathered into a set of construct rules for each. The conditional construct then becomes a tuple that contains the two sets. We summarize this phase in Table 6.6.

Original operator	Rewrite rule
$\dot{C} = (\psi_{\kappa_1\kappa_2...\kappa_n}, \xi)$	$\dot{C} = (V_C, E_C)$, where V_C and E_C are the sets of vertex and edge construct rules returned by Algorithm 3
$\overline{C} = ([\dot{C}_1, \dot{C}_2, ...\dot{C}_n], \lambda_S, \lambda_R, \sigma_S, \sigma_R)$	$\overline{C} = (V_C, E_C)$, where V_C and E_C are the sets of vertex and edge construct rules returned by Algorithm 4. Each conditional construct $\dot{C}_j$ has previously been rewritten.

Table 6.6: Rewrite rules of CONSTRUCT operators.

The process of combining the construct rules of vertices and edges in Algorithm 4 is relatively straightforward. As construct rules contain the variable name of the created vertices or the variable name of the edge and of its endpoints, we can create a mapping between a set of

Algorithm 4: Rewrites a construct clause to a tuple of two sets of construct rules, one for the vertex and one for the edge construct variables in the entire `CONSTRUCT` block. It is expected that the conditional match operators in the input have already been rewritten using Algorithm 3.

input : The construct clause $\overline{C} = ([\dot{C}_1, \dot{C}_2, ...\dot{C}_n], \lambda_S, \lambda_R, \sigma_S, \sigma_R)$, where each conditional construct $\dot{C}_j$ has been rewritten to $\dot{C}_j = (V_{C_j}, E_{C_j})$, a tuple of two sets of construct rules for the vertex and edge construct variables, respectively. Each create rule is a tuple containing the construct variable of the vertex or the construct variables of the edge and incident vertices, a relation from which the new entity can be built and the label and property SET and REMOVE assignments for the new entity.

output: A modified $\overline{C}$ that contains all the vertex and edge construct rules in $[\dot{C}_1, \dot{C}_2, ...\dot{C}_n]$. Construct rules for the same vertex and edge are unioned.

begin
- Create mapping B between a set of variable names and a set of construct rules that contain the respective construct variable(s). B is created over all the construct rules in $[\dot{C}_1, \dot{C}_2, ...\dot{C}_n]$.
- V_C, E_C - the sets of vertex and edge construct variables in $\overline{C}$. Initially, $V_C, E_C \leftarrow \varnothing$.
- **for** set of variable names $b \in B$ **do**
  - $r \leftarrow$ **if** size $B(b) = 1$ **then** $B(b)$ **else** $\bigcup_{c \in B(b)} c$
  - **switch** b **do**
    - **case** vertex variable v **do** $V_C \leftarrow V_C \cup r$
    - **case** endpoints and edge variables a, b, e **do** $E_C \leftarrow E_C \cup r$
- rewrite $\overline{C}$ to tuple (V_C, E_C)
- **return** the new $\overline{C}$

variable names and the construct rules in which the variable(s) appear. Then, for each set of variables that appears in a single construct rule, we add that rule as is to the set of vertex construct rules or to the set of edge construct rules, as per the case. For variables that appear in more than one construct rule, we add the union of all their respective construct rules. The semantics of the union are that the target should first create from each rule the table that will contain the new entity's data (this means evaluating the relation in the rule and add or remove labels and properties, as necessary) and then union these tables. Algorithm 4 has $\mathcal{O}(|Q_C|)$ runtime complexity, where $|Q_C|$ represents the number of construct variables in the construct clause.

6.5 Complexity analysis

In this section we discuss the runtime complexity of the analysis and rewrite phases in the algebraic module in terms of query size. As an overview of all the steps described so far, we present in Figure 6.5 the interpretation pipeline for the query `CONSTRUCT (c {degree_centrality := COUNT(*)}) MATCH (c:Character)-[:HAS_MENTION_WITH]-(:Character)`, in which we add a new property to the Character nodes in the got_graph. We previously used this query as an example in Listing 2.8 when discussing G-CORE's features, in which we also used a graph aggregation in the `CONSTRUCT` clause with the got_graph. Here, we omit the graph union from the query, because we have not yet covered this feature in the interpreter.

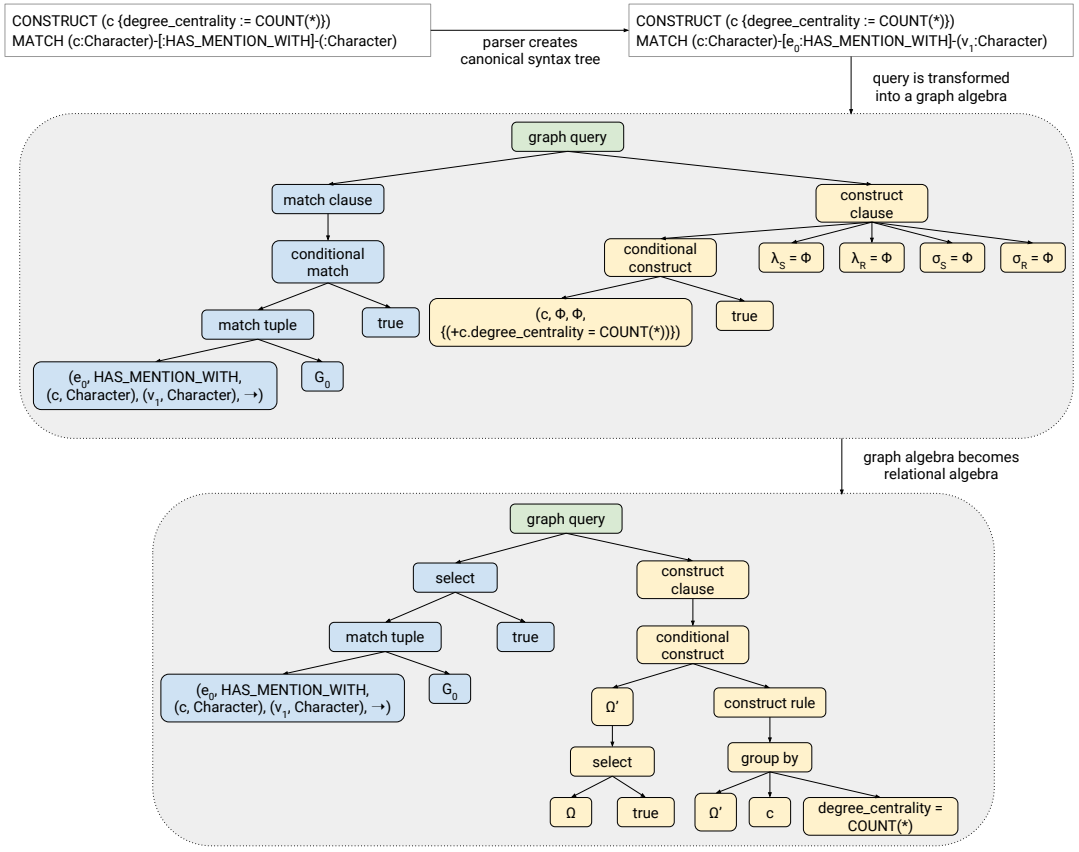

Figure 6.5: Steps taken by the parser and the algebraic module to create a tree of relational operators to solve the query `CONSTRUCT (c degree_centrality := COUNT(*)) MATCH (c:Character )-[:HAS_MENTION_WITH]-(:Character)`.

Sections 6.3 and 6.4 introduced the `MATCH` and `CONSTRUCT` algebra used by the interpreter, as well as the necessary analysis and rewrite phases we needed for transforming the G-CORE-specific operators in the algebraic tree into a combination of relational operators that preserves the language semantics. The algorithms we described are an artifact of the physical and logical data representation we have chosen for the interpreter. By changing this model, the steps we described would, most likely, also need to change.

The rewrite efforts needed by the interpreter are polynomial in terms of query size and/or number of labels in the graph. Let $|Q_M|$ be the number of individual vertex, edge or path patterns in the `MATCH` clause. To infer the labels of each pattern we need $\mathcal{O}(|Q_M| * max(|L_N|, |L_E|, |L_P|)^2)$ steps, therefore the number of labels plays an important role in the rewrite runtime. Once each individual pattern has been annotated with labels, we reshape the `MATCH` sub-tree into its relational equivalent. For this, we will need to split the graph patterns in each match tuple into individual patterns, then rewrite each of the $\mathcal{O}(|Q_M|)$ conditional matches to unions and/or joins - for this we will need $\mathcal{O}(|Q_M|) + |Q_M| * \mathcal{O}(|Q_M|^3) = \mathcal{O}(|Q_M|^4)$. Finally, we apply the selection condition on each conditional match and rewrite the `OPTIONAL` blocks into a left-outer-join, which will have a runtime bound by $\mathcal{O}(|Q_M|)$. In total, the rewrite effort of the `MATCH` block will be $\mathcal{O}(|Q_M| * max(|L_N|, |L_E|, |L_P|)^2) + \mathcal{O}(|Q_M|^4) + \mathcal{O}(|Q_M|) = \mathcal{O}(|Q_M|^4 + |Q_M| * max(|L_N|, |L_E|, |L_P|)^2)$.

The `CONSTRUCT` clause also incurs rewriting overhead. We note that the algorithms used to reshape `CONSTRUCT` were more challenging to write than those used for `MATCH`, i.e. even though the code for `CONSTRUCT` is rather simple, it took considerably more effort to develop algorithms

that preserve G-CORE's semantics of building new graph entities, than it was to find suitable operations on the MATCH block that respect the semantics of the language and our data model. Moreover, it also takes less rewriting effort to translate the G-CORE CONSTRUCT algebra into relational algebra. Let $|Q_C|$ be the number of construct variables in the CONSTRUCT block. The number of conditional constructs is then $\mathcal{O}(|Q_C|)$. We need $\mathcal{O}(|Q_C|)$ steps to generate construct rules from each of the $\mathcal{O}|(Q_C|)$ conditional constructs and $\mathcal{O}(|Q_C|)$ to generate the construct rules for the entire construct clause. In total, the complexity is $\mathcal{O}(|Q_C|^2)$. As a final remark, we note that the CONSTRUCT algebra is simple to rewrite, however, in our implementation, this clause has a considerable runtime complexity in terms of data size, due to the implicit and explicit aggregations of the binding table need by each construct variable (except for unbound and ungrouped vertices, of course) and the joins needed to create the base construct table for edges.

Chapter 7

G-CORE to SQL

In the previous chapters we saw how the G-CORE interpreter parses queries and transforms them into an algebraic tree where the nodes are relational operators. The final step in the interpretation pipeline shown in Figure 4.2 is to translate the query into SQL statements. The module that implements this translation is the target module. Once SQL statements are generated, they are evaluated on the SparkSQL engine to produce DataFrames. GraphX is used for finding shortest paths within the queried graph. In this chapter we discuss how graph data can be made available in the interpreter, how the algebraic tree can be mapped to its SQL equivalent or to a GraphX routine and how we can finally create a new path property graph after we have evaluated the `MATCH` and `CONSTRUCT` sub-trees.

7.1 Importing graph data

Graph data used by the interpreter is stored in DataFrames. We offer the possibility to import it from Parquet[1] or JSON[2] files using a GraphSource - a Scala singleton object that can import graph data from one of the formats. In this case, we require that metadata about the imported graphs be specified through a JSON config file with the format in Listing 7.1. Then graph data can simply be imported as in Listing 7.2. Otherwise, data can be built manually, but it remains necessary to specify the graph name, vertex, edge and stored path data, as well as the edge and path label restrictions (exactly as we do in the JSON configuration above). Once data has been brought into the system, we use the DataFrame schema to create the custom representation of the graph schema required by the interpreter.

7.2 Evaluation of the algebraic tree

The rewriting phases in Sections 6.3 and 6.4 have shown how the G-CORE operators in the algebraic tree are transformed into relational operators and in Section 6.2 we saw that no rewrite rule is applied on the expression sub-trees. In the target module, each algebraic operator is either mapped to its textual SQL representation or triggers the run of a GraphX routine to find shortest paths, the result of which is stored into a DataFrame and subsequently used as a table view in SQL statements. Given the relational operators in the algebraic tree, we are able to create a single SQL query from the `MATCH` sub-tree that evaluates into the binding table, whereas in order to evaluate the `CONSTRUCT` sub-tree we will need to create multiple queries for each of the construct variables.

[1] `https://spark.apache.org/docs/latest/sql-programming-guide.html#parquet-files`
[2] `https://spark.apache.org/docs/latest/sql-programming-guide.html#json-datasets`

```
{
  "graph_name": "some_graph_name",
  "graph_root_dir": "/path/to/graph/data",
  "vertex_labels": ["label1", "label2" ...],
  "edge_labels": ["LABEL1", "LABEL2" ...],
  "path_labels": ["LABEL1", "LABEL2" ...],
  "edge_restrictions": [
    {
      "connection_label": "LABELi",
      "source_label": "labelj",
      "destination_label": "labelk"
    },
    ...
  ],
  "path_restrictions": [
    {
      "connection_label": "LABELi",
      "source_label": "labelj",
      "destination_label": "labelk"
    },
    ...
  ]
}
```

Listing 7.1: Structure of configuration JSON for importing graph data.

```
GraphSource.json(sparkSession).loadGraph("/path/to/config.json")
GraphSource.parquet(sparkSession).loadGraph("/path/to/config.json")
```

Listing 7.2: Importing graph data with a GraphSource stored in JSON or Parquet format.

Creating a SQL query from a part of the algebraic tree means traversing the tree bottom-up and, for each operator, emitting its SQL equivalent. Each node uses the SQL statements generated by its children and combines them into its own SQL representation, which is passed to its parent, and so on. The end result is a single statement that can be run with SparkSQL to create a DataFrame. We exemplify this process in Figure 7.1, in which we generate a query that groups a relation, adds a new column to the result with unique ids and joins the result back to the named relation. There are also special nodes in the algebraic tree that cannot be mapped directly to simple SQL operators and require instead an internal computation. This is the case of `MATCH` and `CONSTRUCT` leaf nodes, which we discuss in the following sections.

7.2.1 Vertex, edge and path scans in the MATCH sub-tree

The leaves of the `MATCH` sub-tree are match tuples in which the graph pattern contains a single vertex, edge or stored path pattern. Conceptually, these patterns are data scans. The evaluation of leaves yields a DataFrame containing an entity's data, which becomes the input of the relational operators higher up in the tree. Tables 7.1, 7.2 and 7.3 provide the SQL templates we use to read vertex, edge and stored path data. We detail the implementation of the three data scan operators in the following paragraphs.

The simplest case for data scan is creating the DataFrame of a vertex - we refer here to simple vertex patterns matched individually in the query, not edge or path endpoints, which are treated differently. The vertex is brought into the system through a projection over all

vertex scan	
algebra	$M = (\varphi_v, G), \varphi_v = [(v, l_v)]$
SQL template	SELECT l_v AS ‘v$table_label‘, {k AS ‘v$k‘, $\forall k \in \theta(l_v)$} FROM l_v

example	`MATCH (c:Character)`
	
	`SELECT "Character" AS ‘c$table_label‘, id AS ‘c$id‘, name AS ‘c$name‘`
	`FROM Character`

Table 7.1: SQL template for reading vertex data.

edge scan	
algebra	$M = (\varphi_{a \xrightarrow{e} b}, G), \varphi_{a \xrightarrow{e} b} = [(e, l_e, (a, l_a), (b, l_b), \rightarrow)]$
SQL template	edge $\leftarrow$ SELECT l_e AS ‘e$table_label‘, {k AS ‘e$k‘, $\forall k \in \theta(l_e)$} FROM l_e src $\leftarrow$ SELECT l_a AS ‘a$table_label‘, {k AS ‘a$k‘, $\forall k \in \theta(l_a)$} FROM l_a dst $\leftarrow$ SELECT l_b AS ‘b$table_label‘, {k AS ‘b$k‘, $\forall k \in \theta(l_b)$} FROM l_b SELECT * FROM (edge) INNER JOIN (src) ON ‘e$src_id‘ = ‘a$id‘ INNER JOIN (dst) ON ‘e$dst_id‘ = ‘b$id‘

example	`MATCH (c:Character)-[e:HAS_ALLEGIANCE_TO]->(h:House)`
	
	`SELECT *`
	`FROM`
	`  (SELECT *`
	`   FROM`
	`     (SELECT "HAS_ALLEGIANCE_TO" AS ‘e$table_label‘,`
	`             id AS ‘e$id‘,`
	`             src_id AS ‘e$src_id‘,`
	`             dst_id AS ‘e$dst_id‘`
	`      FROM HAS_ALLEGIANCE_TO)`
	`   INNER JOIN`
	`     (SELECT "Character" AS ‘c$table_label‘,`
	`             id AS ‘c$id‘,`
	`             name AS ‘c$name‘`
	`      FROM Character) ON ‘e.src_id‘ = ‘c.id‘)`
	`INNER JOIN`
	`  (SELECT "House" AS ‘h$table_label‘,`
	`          id AS ‘h$id‘,`
	`          name AS ‘h$name‘`
	`   FROM House) ON ‘e$dst_id‘ = ‘h$id‘`

Table 7.2: SQL template for reading edge data.

stored path scan	
algebra	$M = (\varphi_{a \overset{@_R}{\Longrightarrow} b}, G), \varphi_{a \overset{@_R}{\Longrightarrow} b} = [(p, rt, l_p, q, (a, l_a), (b, l_b), \rightarrow, \texttt{true}, c, \varnothing)]$
SQL template	path, src, dst computed as for an edge join ← SELECT * FROM (SELECT * FROM (path) INNER JOIN (src) ON ‘p$src_id‘ = ‘a$id‘) INNER JOIN (dst) ON ‘p$dst_id‘ = ‘b$id‘ if rt = `true` ⇒ SELECT ‘a$table_label‘, {‘a$k‘, $\forall k \in \theta(l_a)$}, ‘b$table_label‘, {‘b$k‘, $\forall k \in \theta(l_b)$}[, SIZE(‘p$edge_seq‘) AS c] FROM join if rt = `false` ⇒ SELECT *[, SIZE(‘p$edge_seq‘) AS c] FROM join
example	`MATCH (c:Character)-/@p:CATELYN_TO_DROGO/->(d:Character)`

```
SELECT *
FROM
  (SELECT *
   FROM
     (SELECT "CATELYN_TO_DROGO" AS ‘p$table_label‘,
             id AS ‘p$id‘,
             src_id AS ‘p$src_id‘,
             dst_id AS ‘p$dst_id‘,
             edge_seq AS ‘p$edge_seq‘,
             hops AS ‘p$hops‘
      FROM CATELYN_TO_DROGO)
    INNER JOIN
      (SELECT "Character" AS ‘c$table_label‘,
             id AS ‘c$id‘,
             name AS ‘c$name‘
       FROM Character) ON ‘p.src_id‘ = ‘c.id‘)
INNER JOIN
  (SELECT "Character" AS ‘d$table_label‘,
          id AS ‘d$id‘,
          name AS ‘d$name‘
   FROM Character) ON ‘p$dst_id‘ = ‘d$id‘
```

Table 7.3: SQL template for reading stored path data.

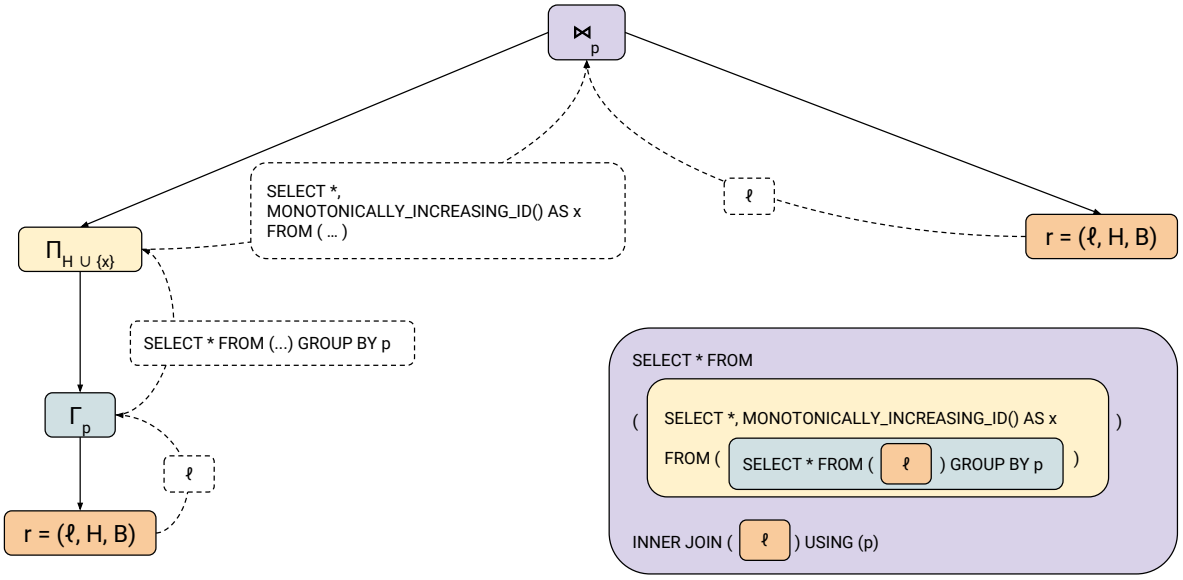

Figure 7.1: Example of how an arbitrary algebraic sub-tree is translated into a single SQL statement. Each node generates its textual representation and passes it to its parent. The query is "collected" at the root.

the columns in the relation stored under the vertex's label name. The projection prefixes each column name with the vertex variable followed by a dollar sign, such that we can later refer to the columns corresponding to the specific vertex when querying the binding table. The renaming is also useful for certain internal analysis performed by the tree nodes before emitting SQL text. Additionally, we also project a column that will contain the label name, a string constant.

The pattern of an edge contains the edge's label, as well as the patterns of its two endpoints. We read the data of the relations stored under the edge's and the two vertices' label names exactly as for a simple vertex. This edge table will contain the two columns src_id and dst_id which reference the unique ids of the two incident vertices. We use these columns to create a new table in which each tuple holds data both for the edge as well as for its endpoints, by joining the edge table with the source and the destination vertex tables. The result is now a single relation that contains the properties of both endpoints and the edge's, as well as a column for each of the three with the label under which they have been matched.

Reading data for a stored path is solved exactly as for an edge. The difference is that a path can be a reachability test, in which case we omit the path data from the result and only include the data of the two endpoints. A cost variable can be defined for a path - in this case we use the SIZE[3] function to determine the length of the edge sequence column and project the result into a separate column.

7.2.2 Computing paths with GraphX

When virtual path patterns (i.e. not stored) are used in the `MATCH` sub-tree, path data can no longer be read from an existing relation and instead the graph needs to be traversed in order to find it. The GraphX framework is a suitable tool for this task and can be easily integrated into the SparkSQL evaluation pipeline, as we show below. At the time of writing this feature is implemented as a proof of concept, in the sense that it treats only a particular query structure - Kleene-star expressions over a given edge label -, but the existing functionality can be reused

[3] `https://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/functions.html#size-org.apache.spark.sql.Column-`

as is, if rewriting logic in the algebraic module is extended. We present a suggestion of how this can be achieved in Section 8.2.

GraphX [10] is a Spark component that supports parallel graph computations and provides an API for well-known graph algorithms. The data model in GraphX is the property graph, which is abstracted through a Graph[VD, ED] class[4] parameterized by VD and ED, the vertex and edge property data types, respectively. This implies two things: (1) there is only one base data type that can be used for vertex properties and only one base data type that can be used for edge properties and (2) multiple properties corresponding to a single graph vertex or edge must be "packed" in a single instance of the property base data type.

A GraphX Graph stores vertices and edges in optimized representations built on top of RDDs. Besides the vertex and edge views, the Graph also offers an *edge triplet view*[5], which is a logical join between the vertex and edge tables. In other words, an edge triplet has access to the edge id and property, as well as to the ids and properties of its incident vertices.

A key feature for our interpreter offered by GraphX is its Pregel operator[6], which can be used to execute graph algorithms iteratively. The Pregel computation is executed in a series of super-steps. In each step, a graph vertex receives messages sent by other vertices in the graph at the previous super-step. The messages are received in bulk. Based on the information contained by the combined messages, the vertex can perform internal computations and transmit its own messages. These messages will be delivered at the next super-step. The vertex computation uses the triplet view to access its neighbors' properties, as well as the incident edges'.

The GraphX API provides a shortest path algorithm[7] that calls into the Pregel operator to find shortest paths between each vertex in a Graph and a sub-set of the graph vertices, denoted landmarks. The algorithm returns a Graph that contains all the vertices in the input, and their property is a mapping between the id of each reachable landmark vertex and an integer representing the number of hops needed to reach the landmark. This algorithm fits our use case, but we need to bring minor modifications to it, such that we can also collect the identifiers of the edges along the path. Another improvement we are interested in is to offer the capability to compute weighted shortest paths, as in its current form the GraphX implementation does not cover this option. To this end, we identify three changes we need to bring to the existing code: (1) we need to convert between our internal data representation and the one used by GraphX in a way that supplies all the necessary information to the shortest paths routine, (2) we need to modify GraphX's shortest paths such that edge ids are collected and the weight of one edge is determined by a supplied cost function, (3) once the Pregel computation has returned the resulting Graph, we need to convert it back to our internal representation.

As mentioned before, we only address a particular query case, in which the path we are computing can traverse any number of edges with a single, particular edge label. Let l_r be the label used in the path regular expression. As per our data model, an edge tagged with the l_r label can only link two vertices of certain labels. Let l_{src} and l_{dst} be two vertex labels, such that $\tau(l_r) = (l_{\text{src}}, l_{\text{dst}})$. We read from the graph storage the DataFrames that hold data for the three labels and we will refer to them as l_r, l_{src} and l_{dst}, respectively. This is the input to our shortest paths algorithm, but it uses our own representation, so we need to convert it into the GraphX model. More specifically, we need to create a single GraphX Graph from the three DataFrames.

To create the GraphX vertices, we only need to provide their ids, which are stored in l_{src} and l_{dst}. We create a single RDD for vertices with the operation $\Pi_{\text{id}}(l_{\text{src}}) \cup \Pi_{\text{id}}(l_{\text{dst}})$, in which we project the id column from each vertex table and union the results into a single relation. To

[4] `https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.Graph`

[5] `https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.EdgeTriplet`

[6] `https://spark.apache.org/docs/latest/graphx-programming-guide.html#pregel-api`

[7] `https://github.com/apache/spark/blob/master/graphx/src/main/scala/org/apache/spark/graphx/lib/ShortestPaths.scala`

Algorithm 5: Pseudo-code to compute the weighted shortest path between all the vertices in a graph and a sub-set of them, called the landmark vertices. The algorithm uses GraphX's Pregel operator.

input : A GraphX Graph G and the set of landmark vertices L. If V is a vertex in G and E an edge, then their property can be accessed with V.prop and E.prop. The property of a vertex is its id, an integer. The property of an edge is a tuple (e_{id}, f), where e_{id} is the edge id and f is its cost function. If E is an edge triplet, then E.src is the source vertex and E.dst is the destination vertex. Each landmark is specified through its id.

output: A GraphX Graph G_{SP}, in which the property of a vertex is a mapping between the id of reachable landmarks and a tuple containing the distance to that landmark and a sequence of edge ids representing the edges that are part of the path to the specific landmark. We alias the map type as the *vertex info map*.

begin
  $G_{SP} \leftarrow \varnothing$
  for vertex $V \in G$ **do**
    $M \leftarrow \varnothing$
    if vertex id $v_{id} \leftarrow V$.prop $\in L$ **then** $M(v_{id}) \leftarrow (0, \varnothing)$
    Add V to G_{SP} with property V.prop $\leftarrow M$
  Add all edges of G to G_{SP}
  return Pregel(graph $\leftarrow G_{SP}$, initialMsg $\leftarrow \varnothing$, activeDirection $\leftarrow$ EdgeDirection.Either)(vprog $\leftarrow$ `VertexProgram`, sendMsg $\leftarrow$ `SendMessage`, mergeMsg $\leftarrow$ `MergeMaps`)

Function `VertexProgram`(vertex id v_{id}, vertex info map P, received info map M)**:**
  return `MergeMaps` (P, M)

Function `SendMessage`(edge triplet E)**:**
  Create M', a vertex info map, as if E's source vertex would follow E to reach the landmarks: $m \leftarrow$ `IncrementMap` (E.prop._1, E.prop._2, E.dst.prop), $M' \leftarrow$ `MergeMaps` (m, E.src.prop)
  if $M' \neq E$.src.prop **then** queue M' to be sent to E.src on the next iteration

Function `IncrementMap`(edge id e_{id_0}, edge cost function f, vertex info map M)**:**
  for vertex id $v_{id} \in M$ mapped to tuple $(d, [e_{id_1}, e_{id_2}, ...])$ **do**
    remap $M(v_{id}) \leftarrow (d + f(), [e_{id_0}, e_{id_1}, e_{id_2}, ...])$
  return M

Function `MergeMaps`(two vertex info maps M_1, M_2)**:**
  $M \leftarrow \varnothing$, the result of map merging
  for each unique key v_{id} in M_1 and M_2 **do**
    $M_x \leftarrow$ either one of M_1 or M_2 that maps v_{id} to the smallest distance
    $M(v_{id}) \leftarrow M_x(v_{id})$
  return M

create the GraphX edges, we only need the ids of the two endpoints - this information is already stored in l_r. As edge property, we will use the edge identifier, such that we can access it to build the path's edge sequence. Therefore, we create an RDD from the edge table with the operation $\Pi_{\text{id,src\_id,dst\_id}}(l_r)$ and, for each RDD row, we create one GraphX Edge[8]. The vertex and edge RDDs are then used to create a Graph, which we denote with G. G is one of the two input arguments of our shortest path algorithm. The other argument is the set of landmark vertices, which we denote with L. We will use a sequence of vertex identifiers to represent L - we can find these identifiers in the id column of the l_{dst} DataFrame.

We have thus converted our internal data representation into G and L, the format accepted by the GraphX framework, so we can now find shortest paths in G. We present the pseudo-code used for this step in Algorithm 5. G and L are the input of the algorithm, however the edges in G also accept a cost function in their property. With the conversion we described above, this cost function is implicitly the constant 1, but more complex functions can be created for weighted paths that, for example, take as argument another edge attribute.

The goal of the algorithm is to create G_{SP}, a Graph that contains all the vertices of G, but with their property changed. We alias the new property type *vertex info map* and it represents a mapping between reachable landmark identifiers and a tuple $(d, [e_{id_1}, e_{id_2}, ...])$, where d represents the cost of the path and $[e_{id_1}, e_{id_2}, ...]$ is the path's sequence of edge identifiers.

We initialize G_{SP} on lines 3-7, by adding to it all the vertices of G with a modified property and preserving the edges as they are. If a vertex is a landmark, then its property is a fresh info map in which the landmark identifier is mapped to the distance 0 and an empty edge sequence. Otherwise, the property of the vertex is an empty info map. We use the Graph's mapVertices function to achieve this, which transforms each vertex attribute in the graph using the map function. The idea behind G_{SP}'s initialization is to start the path finding from the landmarks themselves and sequentially add edges to the path by moving "backwards", towards the source vertex. At the first iteration of the algorithm, the landmark vertices will "know" the distance to themselves is 0 and the other vertices in the graph will have no information about shortest paths.

On line 8, using Scala syntactic sugar, we indirectly call the apply function of GraphX's Pregel operator[9], which takes as input a number of arguments, described in the following one by one. The parameter *graph* is the graph on which the computation will run - in our case, this is the previously initialized G_{SP}. *initialMsg* represents a message that will be delivered to all vertices in the input graph in the first iteration of the computation - we use an empty vertex info map for this.

The *sendMsg* and the *activeDirection* parameters determine which vertices receive messages during an iteration. The activeDirection is set to the value Either[10], which means that the function passed to the sendMsg parameter will be called on both endpoints of an edge if either of the endpoints has received a message at the previous iteration. We supply our custom implementation to the sendMsg function, which follows the template required by the Pregel operator. The argument to this function is the EdgeTriplet that has activated at the previous iteration because one or both of its endpoints have received a message. The sendMsg function is called on a vertex, so the EdgeTriplet argument allows the vertex to have a wider local view - it can access its own property, as well as its incident edge's and the property of its neighboring vertex.

The vertex probes whether by following the edge to its neighbor it can obtain different paths to the landmarks from the ones it already knows. For this, the vertex will first create a possible new info map for itself, by incrementing the info map of its neighbor. This means that for each

[8] `https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.Edge`

[9] `https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.Pregel$`

[10] `https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.EdgeDirection$`

key-value pair in its neighbor's info map, where the key is a landmark identifier and the value is the tuple of distance and edge sequence to that landmark, the key is remapped to the previous distance plus the cost of traversing the linking edge and the edge identifier is prepended to the edge sequence - prepended because the path is essentially built from landmark to path source.

The vertex then tries to merge together its own info map with the one it has just incremented. By merging the two maps, the vertex attempts to update the information it already has about paths with shorter routes obtained by following the edge to its neighbor. The map merging is done by preserving for each unique key in both maps (so for each landmark identifier) the value from either map that had the smallest distance, which means that only the shortest path and its edge sequence will be retained in the merge result.

If the map merging produces an info map different from the one the vertex had - this is the case when either shorter paths have been found or a path through a previously unreachable landmark - the vertex queues the new info map to be sent to *itself* at the next iteration, and because we have set the activeDirection to Either, both endpoints of the edge will by activated and run sendMsg.

To the *vprog* parameter we pass the function that will receive the messages from the previous iteration - in our case, a message can be received by a vertex if that *same* vertex has sent itself a message with fresh information at the previous iteration. For this we supply a function that simply calls the same routine that merges info maps, such that the vertex can update its property with fresh paths.

As an example, suppose there is an edge `(a)-[e]->(b)`, where the vertex `(b)` has discovered a new shorter path at the current iteration. `(b)` will send itself a message, so at the next iteration both `(a)` and `(b)` will run the sendMsg function, which will result in vertex `(a)` discovering its own shorter path going through `(b)` to all the landmarks `(b)` can reach.

Finally, for the *mergeMsg* we supply the same info map merging function, which is called to combine multiple incoming messages into a single info map.

The algorithm will end at the iteration in which no vertex can update its info map in the sendMsg routine. In this case it will simply not queue any more messages to be sent at the next iteration. The Pregel computation converges when there are no remaining messages in the system. There is also a *maxIterations* parameter that can be passed to Pregel, which determines the maximum number of iterations for which the computation should be run, but we leave it to its default value as the maximum integer.

The G_{SP} graph now contains the needed path information, but we still have to convert it back into our internal representation. We need to create a single DataFrame that contains the path data, as well as the data of the incident vertices. Each vertex in G_{SP} has a vertex info map property, in which landmark identifiers are mapped to the distance and edge sequence of the shortest path to that landmark. We create a DataFrame from the vertex RDD in G_{SP} and, to unroll the vertex info map into individual rows we use Spark's explode[11] function. As a final step, this DataFrame is joined with the DataFrames of the source and destination vertices, exactly as we did when reading edge or stored path data.

In Figure 7.2 we show step-by-step how the Pregel-like computation runs on an arbitrary graph and a set of landmark vertices. At the end of the computation the info maps that are the properties of vertices in G_{SP} are transformed into a DataFrame which follows the data layout of paths expected by the target module during the evaluation of the algebraic tree.

[11] `https://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/functions.html#explode-org.apache.spark.sql.Column-)`

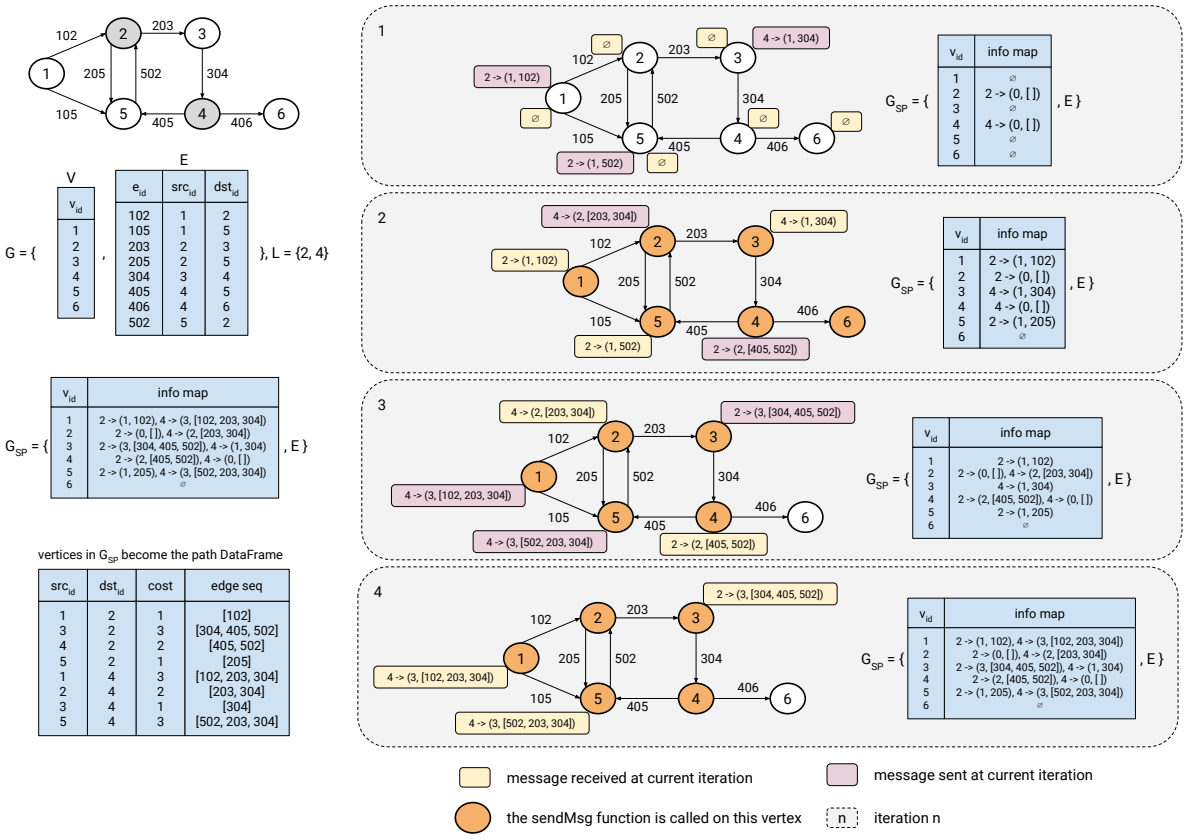

Figure 7.2: Example run of Algorithm 5, in which we look for shortest paths between all the vertices of graph G and the set of landmark vertices $L = \{2, 4\}$. Four iterations of the Pregel-like routine are needed to finish the algorithm. At each iteration, the vertices receive messages from the previous iteration, update their info map property and emit messages to themselves (if it is the case). A vertex is activated in an iteration if it was the recipient of a message at the previous iteration. After shortest paths have been discovered, the vertex info maps in G_{SP} are converted into the representation used by the G-CORE interpreter for paths.

7.2.3 Expressions and canonical relational operators

We translate expression sub-trees recursively into their SQL equivalent. The translation is triggered by the evaluation of algebraic nodes that have an expression as child. For example, the children of a Select operator are a relation and an expression, so when producing the textual SQL representation of that node, the expression is expanded and its text is filled as the argument of WHERE.

In general, expressions have the same syntax in SQL as in G-CORE, with a few exceptions for which we provide the templates in Table 7.4. For a G-CORE `EXISTS` node the graph pattern in the sub-clause has already been rewritten into a relational sub-tree of its own and subsequently into a SQL statement. The sub-clause becomes the SQL EXISTS operator, as discussed in Section 6.2. If the source operator of the outer selection - aliased S - and the graph pattern in `EXISTS` - aliased T - share variables, then we add a condition to the SQL EXISTS that the tuples in S and T have the same identifier. SparkSQL will rewrite the EXISTS into a semi-join.

Another exception to the rule is the group-concat G-CORE expression, which concatenates

expressions	
property reference $x.k$	‘x$k‘
EXISTS φ	EXISTS (SELECT * FROM φ T [WHERE {S.‘x$id‘ = T.‘x$id‘, $\forall x \in H_S \cap H_T$}], where S is an alias for source table of the outer select statement, T is the alias for the table φ is evaluated into, H_S and H_T are the headers of S and T, respectively
power $\xi_L \hat{} \xi_R$	pow(ξ_L, ξ_R)
group-concat([DISTINCT] e)	concat_ws(’,’, collect_list([DISTINCT] e))

Table 7.4: SQL templates for expressions that need a particular translation. The entire expression sub-tree is translated to SQL recursively, using for each node a template. In general, the SQL equivalent of expressions is the same as the G-CORE syntax, with the few exceptions listing above.

a list of strings using comma as the separator. For this we use Spark's concat_ws[12] and collect_list[13] functions.

Table 7.5 provides the SQL templates we used for the canonical relational operators that appear in the algebraic tree. Similar to expressions, their translation is intuitive and the only challenge is to identify correctly the attributes shared between operands, for joins, or missing from either operand, for unions.

7.2.4 Creating new graph entities in the CONSTRUCT sub-tree

We have shown in Algorithm 3 how the conditional construct operators can be rewritten into vertex and edge construct rules. A construct rule contains the construct variable, a tree of relational operators over the binding table - denoted construct relation - and property and label SET and REMOVE assignments for that construct variable. In Section 6.4 we identified three types of vertex construct patterns that are treated differently when creating the construct table for the edges: (1) matched vertices, which are already present in the construct table obtained after solving case 2, and which need grouping by identity to create their own construct rule, (2) unmatched vertices with no explicit GROUP-ing, which we create by simply adding a column with fresh bindings to the filtered binding table and (3) unmatched vertices with explicit GROUP-ing, for which we group the binding table, add a column with fresh bindings and join the result back to the construct table that already contains the vertices from case 1.

Section 6.4 also described the overall flow of creating the new graph entities, in which we can clearly identify two distinct but non-disjoint operations which we need to solve to create the entities. On the one hand, we need to evaluate and materialize the construct relation for each vertex and edge into a DataFrame, because we will use it to add or remove properties and labels to create the final graph entity. On the other hand, we need to create the construct table that contains all the new vertices, such that we have a base table for creating the edges, which need grouping by endpoint and edge identity - and this must be done for *each* conditional construct in the construct clause.

In Figure 7.3 we show an example of how the construct flow is evaluated in the target module for a conditional construct clause. The construction starts from the binding table filtered by the

[12] https://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/functions.html#concat_ws-java.lang.String-org.apache.spark.sql.Column...-

[13] https://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/functions.html#collect_list-java.lang.String-

relational operators	
$r_1 \bowtie r_2$	SELECT * FROM (r_1) INNER JOIN (r_2) USING $(\{a, \forall a \in H_1 \cap H_2\})$
$r_1 \ltimes r_2$	SELECT * FROM (r_1) LEFT OUTER JOIN (r_2) USING $(\{a, \forall a \in H_1 \cap H_2\})$
$r_1 \times r_2$	SELECT * FROM (r_1) CROSS JOIN (r_2)
$\Pi_a(r)$, when $a \notin H$	SELECT {'x$k', $\forall x \in H$ and $k \in \theta(l_x)$}, ROW_NUMBER() OVER (ORDER BY 'a$cid') AS 'a$id' FROM (SELECT *, MONOTONICALLY_INCREASING_ID() AS 'a$cid' FROM (r))
$\Pi_a(r)$, when $a \in H$	SELECT {'a$k', $\forall k \in \theta(l_a)$} FROM (r)
$\sigma_\xi(r)$	SELECT * FROM (r) WHERE ξ
$\Gamma_{k_1,k_2,...;f_1(a_1),f_2(a_2),...}(r)$	SELECT f_1('a$_1$'), f_2('a$_2$'), ..., $\{b, \forall b \in H \setminus \{k_1, k_2, ..., a_1, a_2, ...\}\}$ FROM (r) GROUP BY 'k$_1$', 'k$_2$', ...
$r_1 \cup r_2$	SELECT $\{f(a), \forall a \in H_1 \cup H_2\}$ FROM (r_1) UNION ALL SELECT $\{f(a), \forall a \in H_1 \cup H_2\}$ FROM (r_2), where $f(a)$ = 'a', if a is an attribute in the relation we are selecting from and $f(a)$ = `null` AS 'a', otherwise

Table 7.5: SQL templates for the canonical relational operators. We denote relations with r, r_1, r_2, their headers with H, H_1, H_2 and with l_x the label of variable x.

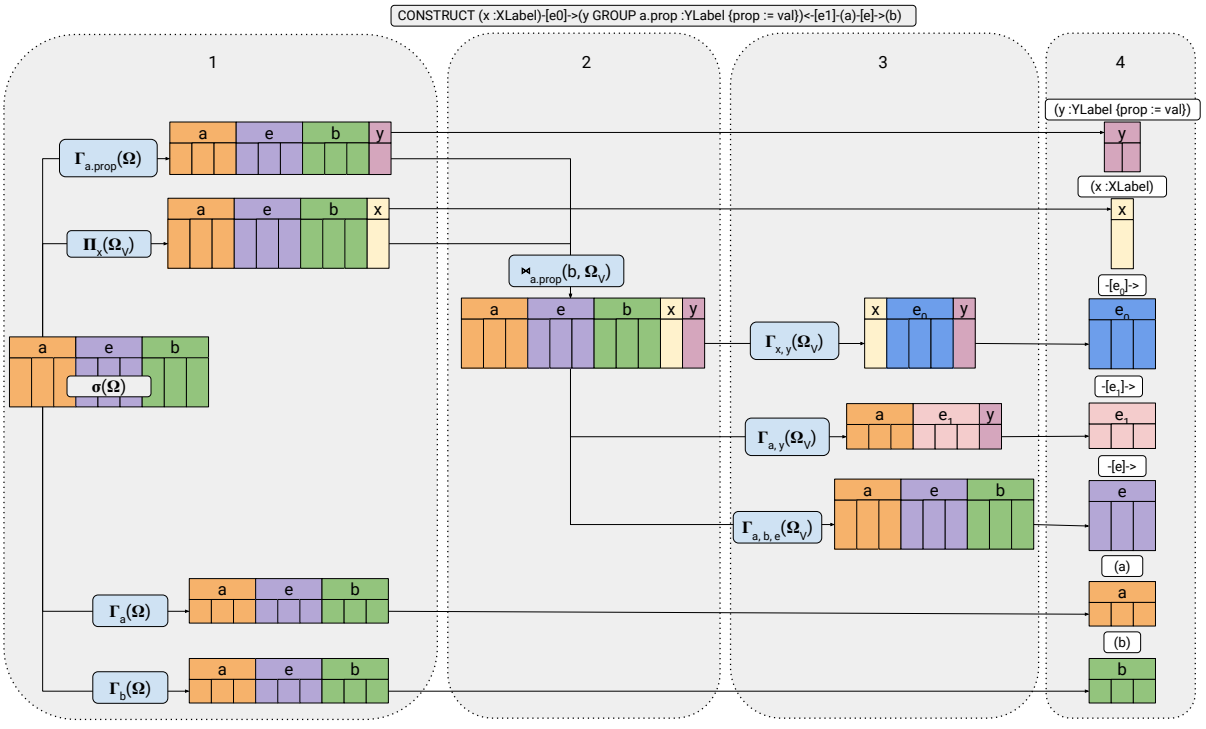

Figure 7.3: Example of construct flow modeled with Algorithm 3 evaluated by the target module for the query `CONSTRUCT (x :XLabel)-[e0]->(y GROUP a.prop :YLabel prop := val)<-[e1]-(a) -[e]->(b) MATCH (a)-[e]->(b)`. The four steps are: (1) create construct rules for vertices, (2) assemble Ω_V, the construct table with vertices, (3) create construct rules for edges from Ω_V and (4) build graph entities from their construct rules.

WHERE sub-clause of the conditional construct. Individual DataFrames are created for each vertex in the conditional construct. These DataFrames will be used in a subsequent step to create the graph entities. The construct table for edges is created in the second step - we start from the DataFrame with the new, unmerged ungrouped vertices and join it with the DataFrames of the new, unmatched grouped vertices. The edge construct relations are then evaluated on the result and it is not necessary to materialize their DataFrames at this step. Finally, to each individual DataFrame, we add and remove properties and labels as instructed by their specific SET and REMOVE assignments.

A final step before creating the graph, which is not shown in the figure, is to rename the columns of DataFrames. We mentioned before that we use a special naming convention for the attributes, by appending the variable name followed by a dollar to the property name. We remove this prefix from the column names before wrapping the data into a PPG instance.

7.2.5 Complexity analysis

We estimate the runtime complexity of the entire query evaluation to be polynomial in terms of data size. The MATCH clause and the individual construct rules are solved by evaluating algebraic trees of relational operators. The operators we use are selection, projection, cross-, inner-, left-outer-, semi-join, union and group by, which have a tractable implementation in SparkSQL. Clearly the evaluation effort depends heavily on the physical implementation of these operators that is chosen at runtime by the Catalyst optimizer, but also on the algebraic tree produced by our rewrite phases and translated to SQL in the target module. In Section 8.1 we identify a possible optimization opportunity for the algebraic plan generated for the MATCH block, that is based on catalog knowledge, i.e. information available to the rewrite phases of the interpreter and that may not be detected by the Catalyst optimizer.

The complexity of the (weighted) shortest paths algorithm presented in this chapter is difficult to estimate due to the convergence condition, however we observe that its implementation is closest to the Floyd-Warshall algorithm, that can find weighted shortest paths in the graph between all pairs of vertices (although our variant is restricted to finding weighted shortest paths to a subset of the vertices, denoted landmarks). The Floyd-Warshall algorithm needs $\mathcal{O}(|V|^3)$ steps to find these paths, where $|V|$ is the number of vertices in the graph. Therefore, it can become quite an intensive computation for large sets of data - which is indeed the case for the very common use-case of social networks. We discuss improvement opportunities for the proposed algorithm in Section 8.3.

Other aspects that should be taken into consideration when analyzing the query complexity are the communication overhead incurred by the computation model in Spark and GraphX, as well as common benchmark metrics, such as memory and CPU utilization. It remains as future work to evaluate the performance of the interpreter based on a comprehensive benchmark set for graph databases, such as the Social Network Benchmark [16].

Chapter 8

Can the interpreter be improved?

In this chapter we discuss a number of steps that could be taken in order to improve the design and implementation we proposed in the previous chapters for the G-CORE interpreter. We briefly sketch ideas that could be used to add more features to the interpreter or to optimize our solutions for features we have already implemented. We also note that these ideas are by no means exhaustive and, as for any piece of software, improvement opportunities can always be found in the existing code.

8.1 Rewrite joins of unions and prune sub-tree

Conditional match operators within the `MATCH` block are rewritten into nested relational operators by way of Algorithm 2. The operators we used were unions, inner-joins and cross-joins: we saw that we union the match tuples that share the entire binding set, then inner-join the operators that share at least one variable, but not the entire binding set, and finally cross-join the remaining non-correlated operators. The result was a left-deep algebraic plan, in which the leaves are the match tuples from the original conditional match. Each of these tuples contains a single pattern of either a vertex, edge or path, which is viewed as a conceptual scan over the data in the graph database - and, therefore, as a table.

A potential problem with this algorithm is that we may end up using the join operator with one or both of the arguments being a union over several tables and use as join key data in columns that, based on catalog knowledge, could never pass the join condition. As an example, suppose we are trying to match the pattern in `MATCH (c1:Character)-[e1]->(c2), (c2)-[e2]->(c3)` on the got_graph in Figure 2.1. The entire block could be satisfied by several combinations of labels, as shown in Figure 8.1, in which we also added a simplified UML diagram of the vertex and edge data schema in the queried graph.

Figure 8.1 also presents the algebraic plan we currently generate, in which we union the tables of each pattern and join the results. Alternatively, we could first join the tables of the two patterns, in which the correlated variable is labeled the same. In our case, the common variable between the two patterns is `(c2)`, the vertex that is the destination of the first edge pattern and the source of the second edge pattern. The results of the union could then be joined.

As a generic example, suppose we are solving two patterns that are correlated through one variable. We believe it is worth investigating whether rewriting joins of unions to unions of joins could lower the computation time. All the property keys of the correlated variable are used as join key, each label has its own set of property keys and we use one table per label in the graph storage. As we join on the correlated variable, when this variable is labeled differently in the two patterns, we can know for sure that there cannot possibly be any tuple in either pattern that would pass the join condition. Instead, if the correlated variable is labeled the same on both sides of the join, it is possible to find matching tuples.

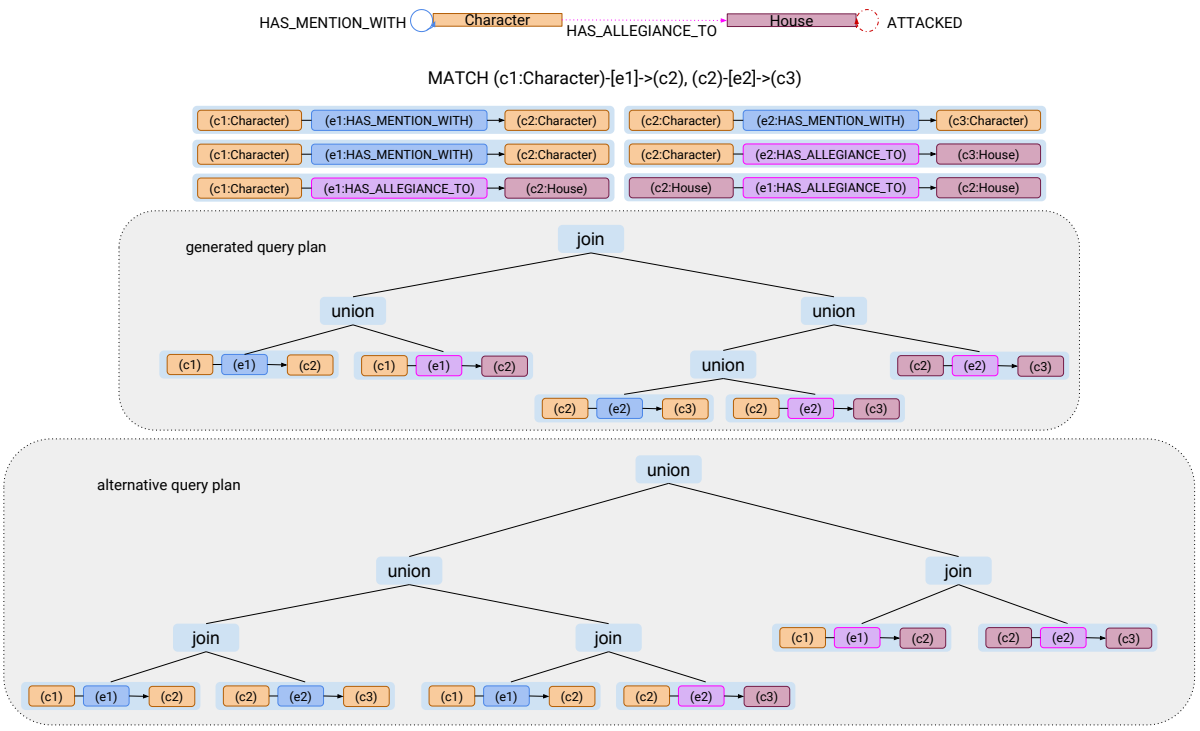

Figure 8.1: The various ways in which the block `MATCH (c1:Character)-[e1]->(c2), (c2)-[e2]->(c3)` could be annotated with labels by the label inference algorithm, the algebraic plan we currently generate and an alternative algebraic plan for the same query.

As a simple implementation of this rewrite phase, we could first generate the plan we are already doing and then use rewrite logic to reshape it, if it proves worth changing the evaluation flow. Whenever a pattern of join of two union sub-trees is encountered in the algebraic tree, the rewrite phase could generate all combinations of joins between all the leaves under the left and right arguments of the join. We could then prune those joins in which the correlated variable would be labeled differently on the two sides.

An estimation of the time complexity of the two flavors of the algebraic plan depends heavily on the physical plan that is chosen at runtime by the underlying system - SparkSQL, in our case. Suppose both the first and the second pattern can be labeled in k ways. Also, let's assume that each of the k tables of a pattern has n elements. We are interested in estimating the evaluation complexity of the two algebraic plan flavors in terms of data size. We will use $j(n)$ to denote the complexity of the physical join implementation and $u(n)$ to denote the complexity of the physical union implementation. j and u are placeholders for their respective big-O notations. In the first case, in which we join the union of the two patterns, we would need $k * u(n) + j(k * n)$ time to evaluate the algebraic plan, where $k * u(n)$ is the time needed to union the tables of both patterns and $j(k * n)$ is the time needed to join the two unions.

When instead using union over smaller joins, we run into another factor that contributes to the runtime complexity, namely the number of joins that remain after the pruning. For example, the correlated variable could be labeled in k different ways in the first pattern and k different ways in the second pattern, having a one-to-one match between the two sides. In this case, we would only generate k joins. However, we could also have the case in which the correlated variable in labeled k times the same way in both patterns. Then we would generate k^2 joins, because none of them would get pruned. Let $f(k)$ be the number of joins that remain in the algebraic tree after pruning - it could be $\mathcal{O}(k)$ or $\mathcal{O}(k^2)$. Then the runtime complexity of the rewritten algebraic plan would be $f(k) * j(n) + f(k) * u(n)$, where $f(k) * j(n)$ is the time needed

to evaluate the $f(k)$ joins and $f(k) * u(n)$ the time needed to evaluate the union of the $f(k)$ results of the join operations.

We can see that there are several factors that contribute to the evaluation runtime of the MATCH block - the complexity of the join and union implementations that are used in the physical plan, as well as the number of joins we would be generating after this rewrite phase. The two approaches should be compared on a comprehensive benchmark set that closely follows real-world workloads, in order to identify whether it would make sense in practice to use more, but smaller joins, instead of a single, bigger operation.

8.2 Rewrite the PATH clause with sub-queries

Weighted shortest paths in a G-CORE query can be specified with the help of PATH patterns. While a syntactically valid query must have one CONSTRUCT and one MATCH clause, any number of optional PATH blocks can be used to define a complex structure for the atoms of a matched path and/or to specify a weight function for each atom. The structure of the PATH clause, presented in Listing 8.1, specifies a graph pattern to be used for matching the path atoms (or segments) within the graph, with the possibility to also filter bindings by the WHERE condition. The entire pattern is aliased with a path macro name. The COST sub-clause is also optional and can be used to provide a cost function for the matched atoms. The design of G-CORE [20] mentions that the first and last nodes in the graph pattern must be the start and end nodes of a path segment.

```
PATH <path_macro> = <graph_pattern> WHERE <condition> COST <cost_def>
```

Listing 8.1: Structure of the PATH clause in G-CORE.

After the path structure has been defined and aliased, we can use its macro name in subsequent PATH clauses to create more complex path patterns, or in the MATCH clause, in which case the semantics are to use the graph patterns of the path segments with their associated cost function to search for weighted shortest paths in the graph, that follow the predefined structure of the path atoms.

The implementation of the interpreter, as presented in this thesis, did not cover PATH structures. In Listing 8.2 we outline an idea on how sub-queries could be used to rewrite a PATH clause before matching its expression. Given a PATH clause, which has a start and stop vertex for its segment pattern, we propose to rewrite the clause to a sub-query in which we match the path pattern as is, along with its WHERE condition. For each binding we then add a new edge in the graph between the start and stop nodes, with a certain label (for example, it could be the macro name). The cost of the segment can be filled in as a property to this new edge. In the outer query we can then use MATCH to look for shortest paths in the graph that only contain the newly added edges.

```
PATH macro_name = (start)- ... ->(stop) WHERE cond COST cost_def
CONSTRUCT ...
MATCH (a)-/<~macro_name*>/->(b) ON g

CONSTRUCT ...
MATCH (a)-/<:MACRO_LABEL*>/->(b) ON (
  CONSTRUCT g, (start)-[:MACRO_LABEL {cost := cost_def}]->(stop)
  MATCH (start)- ... ->(stop) WHERE cond ON g
)
```

Listing 8.2: Rewrite rules for PATH clauses. Top: a G-CORE query using a PATH clause. Down: the query is rewritten such that we first match the path pattern on the graph and add a new edge between the start and stop node of the pattern, in a sub-query. The outer query then tries to find shortest paths that traverse the newly added edges.

A simple example of the rewrite rule is shown in Figure 8.2. The edges in the graph are labeled with the color they are drawn - GREEN and ORANGE, respectively. The `PATH` clause defines a pattern that traverses one GREEN edge followed by one ORANGE edge. We add a new BLUE edge in the graph between the start and stop node of each of the segments that satisfy the path pattern and then simply query for shortest paths over BLUE edges.

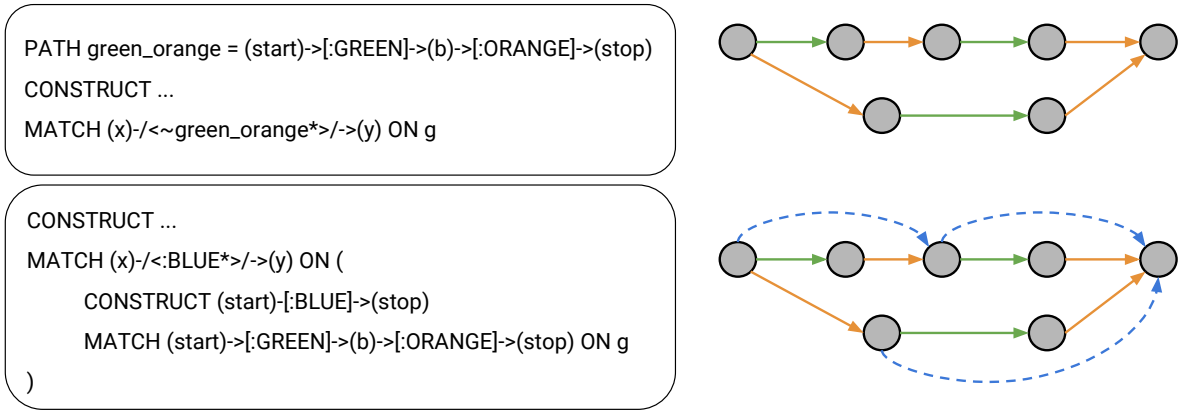

Figure 8.2: Example of a G-CORE query using a `PATH` clause. Each edge is labeled with the color it is drawn. Top: the `PATH` clause is used to define a path segment containing vertices linked by one GREEN edge followed by one ORANGE edge. Down: the query is rewritten such that we first match the path pattern and for each binding we add a BLUE edge between the start and end node of the pattern in a sub-query. The outer query then tries to match the shortest paths over BLUE edges.

If such a rewriting mechanism is used, then a few changes would need to be brought to the current query solving mechanism. First, the default cost of matched patterns is the number of hops in the path. When `PATH`s with a `COST` function are rewritten to an edge with the result of the function filled in as an edge property, the interpreter should propagate this information to the path algorithm. For example, this could be accomplished by providing an extra constructor parameter to the leaf node that represents virtual paths, in which we could encode the name of the cost property. Another solution could be to use a specific name for the cost property and to pass a boolean to the leaf, which tells whether that property should be used or not, or simply use it, if it is present in the schema of the new edge label DataFrame. The second change we would need is to find a mechanism to remove the edges we would add for each path segment, before returning the result of the outer query. A possible solution would be to use a specific label or label prefix, which instructs the interpreter to not add the DataFrames that contain the new edges to the resulting graph.

8.3 Discussion on shortest paths algorithms

Section 7.2.2 presented a rather naive algorithm for computing shortest path between all the vertices of a graph and a subset of them, called the landmark vertices. For this purpose we used a GraphX algorithm that uses a Pregel-like computation to determine the cost and the edge sequence of shortest paths. We can identify a number of improvements that can be brought to this particular algorithm, but also to the entire solution we suggested for shortest paths.

The first problem we pinpoint is that our solution uses a one-size-fits-all approach and does not consider the actual expanse of the problem. An ideal solution should first create the bindings for the vertices that participate in the algorithm by applying all the filters specified for the vertices and edges of the path pattern. This ensures that we run the computation on the smallest

possible input that satisfies the constraints of the query. The next step would be to choose a variant of the shortest paths algorithm using the problem size as heuristic. A good start would be to differentiate between the single-source, single-pair and all-pairs flavors of the task. Rutgers presents and compares in his Master's thesis [32] a number of algorithms for weighted shortest paths running on the Lighthouse graph engine, which leverages the Apache Giraph framework [1] to run graph queries in a distributed fashion. Both Giraph and GraphX use the vertex-centric computational model of Pregel [28] to express graph algorithms as a series of iterations, which makes his thesis an interesting read for the implementation of path algorithms in the G-CORE interpreter.

The second observation we make is that the size of the vertex attribute in the proposed GraphX solution should also become a parameter of the problem. The output of the computation should be not only the cost of the path, but also the identifiers of the edges along the path, in the order they occur in the path. Our solution was to build for each vertex the *vertex info map*, a mapping between landmark identifier and the list of edge identifiers in the path. The problem is that these *info maps* can grow arbitrarily in size, thus affecting the memory footprint, communication and computation overhead of the algorithm. This is also an issue identified by Rutgers in [32], for which he proposes a number of optimizations in his thesis. We believe the impact of his ideas on the query evaluation time in the G-CORE interpreter could be analyzed and, if proven useful, adopted.

Chapter 9

Can G-CORE be improved?

In this chapter we evaluate G-CORE from the perspective of a user and then assess the semantics of the language, using insights we gained by implementing an interpretation pipeline for G-CORE queries. While developing the G-CORE interpreter we extensively used the formal semantics presented in the initial version of the paper, available in the arXiv repository [21]. Another useful tool for developing the interpreter was the guided tour of G-CORE provided in its design [20], which highlights and explains the main features of the language through examples. We make the commentary that while the semantics in [21] are useful for formal proofs and the more relaxed introductory examples in both versions of the paper [21, 20] are a gentle introduction to the syntax, a more appropriate tool for G-CORE users would be to synthesize the language features in a colloquial, but structured and comprehensive definition as the ones PGQL [14] and openCypher [5] offer.

Our general impression of G-CORE is that its multitude of features have been very well thought through and the result is a comprehensive language set that can be used in practice for path property graph databases. During our work we have not encountered queries that we could not express with G-CORE, though we must remark that our experience with the language is limited and only an extensive use in practice could surface incomplete or missing features.

9.1 A user's perspective

While working on the G-CORE interpreter and on this thesis we wrote a number of G-CORE queries with the purpose of unit-testing the code, hand-testing the interpreter or giving the examples in this thesis. As users of the new language we pinpoint and discuss a number of features we found ambiguous or that we believe could be changed to improve the user experience with the language.

9.1.1 MATCH before CONSTRUCT

The first issue we note as users of the G-CORE language is the order in which the two G-CORE clause `MATCH` and `CONSTRUCT` appear in a query. While writing examples, we noticed that we would always start writing the query from the `MATCH` clause, then followed by the `CONSTRUCT` statement. This tendency to first think of the patterns we want to match in the graph and then to use them to build the result is also reflected in the evaluation semantics, as the construct patterns are evaluated in the presence of the binding table created by `MATCH`.

Our opinion is that the order of the three blocks in a graph query should be `PATH`, followed by `MATCH`, followed by `CONSTRUCT`. Another option could be to allow both variations, `CONSTRUCT` followed by `MATCH`, as it is now, or the one we propose, `MATCH` followed by `CONSTRUCT`, such that users could choose the one that makes more sense to them. It can be argued that by introducing

this change G-CORE will lose from its declarative nature, as the order of the clauses in the query would "tell" the interpreter "how to evaluate" the query, and not "what to evaluate". However, as the query does not return tabular data, `CONSTRUCT` is not a simple SQL SELECT statement - it has more complex semantics and at times necessitates reasoning about the data in the binding table. Thus, exactly as the query evaluation flow, as users we first designed the `MATCH` statement of our queries and only then used a "mental view" of the binding table to add the `CONSTRUCT` patterns.

9.1.2 Not always desirable to return graphs

While it can be handy to use a full G-CORE query that matches graph patterns and builds a graph from the result, we observe that building a new graph is costly in terms of evaluation time, because of the aggregations and joins on the binding table that are used in the process. This issue is also addressed in the G-CORE paper [20], in which a SQL extension for the language is proposed, such that tabular projections could be used on the results of the `MATCH` clause, or tabular data could be the input of either `MATCH` or `CONSTRUCT`. For nested queries we clearly need a mechanism to pass graphs from the inner to the outer queries, or an equivalent representation of the graph data supported by the language. Another situation in which graph construction/aggregation is useful is, for example, path discovery - as this is an expensive operation, it can be done once and then have the new paths stored back to the graph, such that subsequent queries could simply search for the previous results. However, there are cases when we might not be interested in running the entire evaluation pipeline that creates graphs - examples would be computing graph metrics or mining graph data. In these situations, support for tabular data is appropriate.

9.1.3 ON can be ambiguous

During our work on the interpreter implementation we also noticed that the `MATCH ... ON ...` syntax can be ambiguous, due to the existence of the default graph feature in G-CORE. For example, our first impression as a user was that in the query `MATCH (a)->(b)->(c), (a)->(d) ON g` the two chained patterns `(a)->(b)->(c)` and `(a)->(d)` would both be matched on the graph `g`. However, this is not the case. As per G-CORE's open-source grammar [7] and, implicitly as per the algebra we designed in Chapter 6, the two patterns become a match tuple each, in which the first pattern is matched on the default graph (as its `ON` sub-clause is not specified), while the second is matched on graph `g`. A semantic conflict arises when `g` has not been defined as the default graph in the system, making the query ambiguous. This is clearly a problem that can be flagged in the interpreter as an exception, or a "graph inference" mechanism could be implemented to annotate patterns using the default graph with the correct named graph, if a correlated pattern uses one. However, this makes the language less transparent to the user. Using the entire pattern before the `ON` token (therefore also the comma-separated sub-patterns) to create a match tuple could also be a solution, but this approach has its own downsides, as it removes the default graph feature and requires the user to always specify on which graph to match.

9.2 Assessing G-CORE's formal definition

In this section we report a number of ambiguities we discovered in the the language definition and semantics - they are exclusively related to the `CONSTRUCT` clause. We remark that the implementation of the evaluation pipeline was straightforward for the evaluation of the `MATCH` clause, while the `CONSTRUCT` block proved to be more complicated because of the need to solve

the construct rules and the operations on the binding table in the right order, such that we cover all use-cases and preserve the semantics of the language.

9.2.1 Is WHEN a pre- or post-aggregation condition?

The first question we ask is whether the WHEN sub-clause is a condition that should be applied on the binding table before or after its aggregation. The CONSTRUCT clause uses implicit grouping on the binding table for the variables that have been matched and, additionally, explicit aggregations can be used with the help of the GROUP statement. G-CORE's design presents examples of queries that use WHEN both for filtering before aggregation and for filtering after aggregation. We present the two examples in Listings 9.1 and 9.2.

```
CONSTRUCT social_graph, (x GROUP e :Company {name :=e})<-[y:worksAt]-(n)
  WHEN exists(e)
MATCH (n:Person {employer = e})
```

Listing 9.1: Listing lines 20-23 from [20]

```
CONSTRUCT (n)-[e:wagnerFriend {score := COUNT(*)}]->(m)
  WHEN e.score > 0
MATCH (n:Person)-/@p:toWagner/->(), (m:Person) ON social_graph2
  WHERE m = nodes(p)[2]
```

Listing 9.2: Listing lines 68-72 from [20]

In Listing 9.1 the query matches nodes labeled Person in the graph and unrolls the multi-valued property "employer" into the binding `e`. In CONSTRUCT a new node `x` is created WHEN the unrolled property `e` is non-null. Clearly, this is a pre-aggregation filtering. In Listing 9.2, in the construct pattern of the new edges `e`, we add to each edge the new aggregated property "score", and create an edge WHEN this score is greater than zero. This filtering is therefore applied on the grouped binding table.

The formal semantics in [21] mention that the evaluation of a WHEN expression can use both variables from the binding table, as well as the new variables in construct patterns. The expression is evaluated in the presence of the binding table joined with the new bindings of the construct pattern. This means that the evaluation order would be: group the binding table as is and in the same time create the aggregated properties, join back to the binding table to add the new properties and variables to it, filter out the bindings - both old and new - that do not satisfy the WHEN condition. Notice that this would mean that in Listing 9.1 we would also group by the null key. We observe that having both pre- and post-aggregation conditions in the same expression tree increases the size of the data that is aggregated and joined or the complexity of the interpreter. We believe it would be more feasible and easier to implement G-CORE if the two types of conditions would be separated as in SQL's WHERE and HAVING and we would evaluate the two in different parts of the interpretation pipeline.

9.2.2 Ambiguous conditions in WHEN

A second ambiguity we found with WHEN were conditions applied on chained construct patterns as the one in Listing 9.3

```
CONSTRUCT (a)-[e1]->(b)-[e2]->(c)-[e3]->(d)
  WHEN SUM(x.prop) >= 42
MATCH (b)-[e2]->(c), (x)
```

Listing 9.3: Ambiguous condition in the WHEN clause.

Let's assume Table 9.1 is the binding table to which MATCH evaluates. For simplicity, we identify b, e2 and c by their ids and x by the value of x.prop.

b	e2	c	x
10	20	30	40
10	20	30	2

Table 9.1: We assume the MATCH clause in Listing 9.3 evaluates to this binding table.

The construction of the entire graph pattern starts with building the vertices. The vertex variables a and d are not bound by the MATCH block, therefore they will receive consecutive identifiers, padded as new columns to the binding table. The result is the one in Table 9.2, in which we use the ids of a and d as the vertex identity. Already we can ask, does the WHEN condition play any role when building the two new vertices? There is no aggregation involved in this process, however the condition is on an aggregated property.

a	b	e2	c	d	x
50	10	20	30	60	40
51	10	20	30	61	2

Table 9.2: The new variables a and d receive two distinct fresh bindings.

The next step in solving the chained construct pattern is to create the edges. Each new edge requires an aggregation of the binding table, therefore, which should the WHEN condition be applied on? For e2 we use b, c, e2 as the aggregation key. In this case, the sum of x's property is $40 + 2 = 42$, which satisfies the condition. However, for e1 and e3 we group by a, b and c, d, respectively. Each of the groupings creates two buckets, for which the WHEN condition is not satisfied. The question is, should we build any vertex or edge at all? Should we only build e2 and its two endpoints?

9.2.3 Does CONSTRUCT impose an order for variable construction?

In Chapter 2 we provided examples of G-CORE queries to highlight and explain its main features. Listing 2.3 made us ask: is there an implicit order for evaluating the list of conditional constructs in the CONSTRUCT clause? For a quick reference, we provide the part of the query we are interested in, in Listing 9.4.

```
CONSTRUCT
  (b GROUP a.battle_name :Battle {name := a.battle_name}),
  (h)-[:WAS_IN {role := "attacker"}]->(b) WHEN (h)-[a]->(),
  (h)-[:WAS_IN {role := "defender"}]->(b) WHEN (h)<-[a]-()
MATCH (h:House)-[a:ATTACKED]-()
```

Listing 9.4: Is there an implicit order in the construct clause?

In this example, the interpreter would need to first build node b with its explicit GROUP declaration and only after this operation proceed with creating the edges in the subsequent conditional constructs. When building the edges, we would need to use the binding table to which the new bs would have been added in the previous step, which creates new semantics for evaluating conditional construct clauses correlated to other conditional constructs - the correlated clause is not evaluated on the binding table, but instead on the binding table joined with the bindings created for construct variables that are built in the correlated clauses. However, how can we determine which conditional construct to evaluate first? In our case, we conveniently

place `(b GROUP ...)` as the first conditional construct in the construct clause, which syntactically makes more sense, but this may not always be the case. Moreover, what happens when correlated conditional constructs use conflicting `WHEN` conditions?

9.2.4 What are the semantics of graph patterns in WHEN?

In the previous Listing 9.4 we also used graph patterns in the `WHEN` condition. What are evaluation semantics in this case? Do they follow the same rules as for the existential sub-queries that replace graph patterns in `WHERE`? If this is the case, then a G-CORE interpreter would need an additional step in which it annotates each pattern in the `WHEN` condition with the default graph or a named graph determined by correlated patterns in the `MATCH` clause.

Chapter 10

Conclusions

G-CORE [20] is a new graph query language for path property graph databases. It has been designed by members of the academia and industry with the purpose of creating an archetype graph query language and to this end the most useful and desirable features of existing languages have been captured into the syntax and semantics of G-CORE. Moreover, new features have been embedded into the language: queries always return graphs, which makes it G-CORE a composable language, and paths have become first-class citizens in the graph, by being part of the data model.

Graph queries in G-CORE must use the two main clauses `MATCH` and `CONSTRUCT`. The `MATCH` block is used for graph pattern matching and is evaluated into a maximal set of bindings for all the variables in the clause, that satisfy all the graph patterns in the block. The resulting binding set is then used by `CONSTRUCT` to create a new graph, based on the construct patterns provided in the query. As paths are part of the data model, they can be stored and later queried. The G-CORE syntax and semantics support path queries in the `MATCH` clause, where we can search for top-N shortest paths, all paths or determine whether two nodes are reachable through a path. Weighted shortest paths are also supported by way of the `PATH` clause, in which complex graph patterns can be used to define the structure of the path segments and/or a cost function can be defined for each segment.

In this thesis we implemented an interpreter that is able to evaluate G-CORE queries, by performing pattern matching on the graph database and creating new graphs from the matched data. We followed the language definition and formal semantics of the language presented in the first version of G-CORE's design [21] and in the published paper [20]. Our research questions were presented in Section 1.1 and our main goal was to assess G-CORE's design through the implementation of an interpreter and to verify whether we can find a suitable data representation and tractable algorithms to solve G-CORE queries. We wrote the interpreter in Scala and decided from the beginning to use a relational model for the physical data, as well as for representing the binding table. Coupled with the translation of G-CORE queries into SQL statements, this allowed us to leverage the functionality of an existing relational database system. We chose to use Spark DataFrames for the graph data and SparkSQL as the execution engine of our interpreter. These design decisions have been presented in Chapter 4.

In order to solve G-CORE queries, we needed to pass the query through several analysis and rewrite phases, detailed in Chapters 5, 6 and 7. We represented the query internally with the help of a graph algebra tailored for the G-CORE queries. We defined the `MATCH` and `CONSTRUCT` algebra in Chapter 6. In subsequent steps, we transformed the graph algebra into a relational algebra that uses the standard operators selection, projection, cross-, inner-, left-outer- and semi-join, union and aggregation. We determined the runtime complexity of the rewrite efforts to be polynomial in terms of query size and number of labels in our graph database, with perhaps a more considerable effort put into the so-called "label inference" algorithm.

We discussed the process of translating the relational algebra into SQL statements in Chapter 7. We determined the runtime complexity to be polynomial in terms of data size, because the evaluation of the relational operators we used is tractable. To search for paths in the graph, we adapted an existing GraphX algorithm that finds paths between all the vertices in the graph and a sub-set of nodes, denoted landmarks. The algorithm we propose only covers a particular use-case of path finding and has been implemented as a proof-of-concept - we showed that path patterns can be used in the `MATCH` clause and that there are available tools and algorithms that can be adapted to our data and computation model. The runtime complexity, though polynomial, can become considerable for large sets of data. Improvements for this algorithm, as well as for the implementation of other features have been discussed in Chapter 8.

We conclude that we did manage to find suitable algebraic primitives to solve G-CORE queries in polynomial time and that G-CORE is a rich language, with which we managed to write a multitude of queries, that we used to evaluate G-CORE's expressiveness. We did encountered a number of ambiguities in the semantics of the `CONSTRUCT` clause, which we discussed in Chapter 9. Overall we believe G-CORE is a suitable graph query languages for path property graph databases and, with some improvements, it can indeed be a point of reference for future graph query languages or for a language standard.

Bibliography

[1] Apache Giraph. http://giraph.apache.org/. Accessed: 2018-06-06.

[2] Apache Spark: Lightning-fast unified analytics engine. https://spark.apache.org/. Accessed: 2018-06-11.

[3] Apache TinkerPop. http://tinkerpop.apache.org/. Accessed: 2018-06-06.

[4] CAPS: Cypher for Apache Spark. https://github.com/opencypher/cypher-for-apache-spark. Accessed: 2018-06-11.

[5] Cypher Query Language Reference, Version 9. https://github.com/opencypher/openCypher/blob/master/docs/openCypher9.pdf. Accessed: 2018-06-06.

[6] Datastax. The Benefits of the Gremlin Graph Traversal Machine. https://www.datastax.com/dev/blog/the-benefits-of-the-gremlin-graph-traversal-machine, note = Accessed: 2018-06-12.

[7] G-CORE Grammar and Parser. https://github.com/ldbc/ldbc_gcore_parser. Accessed: 2018-06-07.

[8] Game of Thrones. Explore deaths and battles from this fantasy world. https://www.kaggle.com/mylesoneill/game-of-thrones/data. Accessed: 2018-06-05.

[9] GraphFrames User Guide. https://graphframes.github.io/user-guide.html. Accessed: 2018-06-12.

[10] GraphX | Apache Spark. https://spark.apache.org/docs/latest/graphx-programming-guide.html. Accessed: 2018-07-10.

[11] Intro to Cypher. https://neo4j.com/developer/cypher-query-language/. Accessed: 2018-06-11.

[12] mneedham/neo4j-got. https://github.com/mneedham/neo4j-got/tree/master/data/import. Accessed: 2018-06-05.

[13] MonetDB. https://www.monetdb.org/Home. Accessed: 2018-06-15.

[14] PGQL 1.1 Specification. http://pgql-lang.org/spec/1.1/. Accessed: 2018-06-12.

[15] PGQL · Property Graph Query Language. http://pgql-lang.org/. Accessed: 2018-06-06.

[16] Social Network Benchmark | LDBCouncil. http://ldbcouncil.org/developer/snb. Accessed: 2018-08-01.

[17] The Neo4j Graph Platform. https://neo4j.com/. Accessed: 2018-06-06.

[18] The openCypher Project. https://www.opencypher.org/. Accessed: 2018-06-06.

[19] The Spoofax Language Workbench. `http://www.metaborg.org/en/latest/index.html`. Accessed: 2018-06-19.

[20] R. Angles, M. Arenas, P. Barcelo, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, and H. Voigt. G-core: A core for future graph query languages. pages 1421–1432, 2018.

[21] R. Angles, M. Arenas, P. Barceló, P. Boncz, G. H. L. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, and H. Voigt. G-CORE: A Core for Future Graph Query Languages. *ArXiv e-prints*, Dec. 2017.

[22] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, Sept. 2017.

[23] R. Angles and C. Gutierrez. Survey of Graph Database Models. *ACM Comput. Surv.*, 40(1):1:1–1:39, Feb. 2008.

[24] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[25] A. Beveridge and J. Shan. Network of Thrones. *Math Horizons Magazine*, 23(4):18–22, 2016.

[26] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1433–1445, New York, NY, USA, 2018. ACM.

[27] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association.

[28] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[29] M. E. J. Newman. The structure and function of complex networks. *SIAM REVIEW*, 45:167–256, 2003.

[30] M. A. Rodriguez. The Gremlin Graph Traversal Machine and Language. *CoRR*, abs/1508.03843, 2015.

[31] N. P. Roth, V. Trigonakis, S. Hong, H. Chafi, A. Potter, B. Motik, and I. Horrocks. PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, GRADES'17, pages 7:1–7:6, New York, NY, USA, 2017. ACM.

[32] P. Rutgers. Extending the Lighthouse graph engine for shortest path queries. Master's thesis, 2015.

[33] M. Sevenich, S. Hong, O. van Rest, Z. Wu, J. Banerjee, and H. Chafi. Using domain-specific languages for analytic graph databases. *Proc. VLDB Endow.*, 9(13):1257–1268, Sept. 2016.

[34] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: A Property Graph Query Language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, pages 7:1–7:6, New York, NY, USA, 2016. ACM.