Graph Database & Query Evaluation Terminology

Introduction to Database & Querying Concepts

Roan Hofland
roan@roanh.dev

23rd of September, 2024

Contents

Version History

Version	Date	Comment
v1.0	2024-09-09	Initial version.
v1.1	2024-09-16	Operation figures and language section extensions.
v1.2	2024-09-16	Small improvements and AST figure.
v1.3	2024-09-23	Minor clarifications and corrections.

1 Introduction

This report is intended as a short introduction to the terminology and concepts used in databases and query evaluation. Although the focus of this report will be on graph databases, it is worth noting that most concepts apply to relational databases as well. The main purpose of this report is to give a condensed easy to reference overview to serve as a basis when working with gMark [6]. Note that most concepts introduced have more in-depth explanations in my thesis on *CPQ*-native Indexing [7] and my other prior research [5, 4, 8], in fact, most of the sections in this report are loosely based on existing content from those research items. For those more interested in the lower level details of how graph databases are implemented, the "Querying Graphs" book is also highly recommended [3], as well as the more general "Database System Concepts" book [10].

2 Database Concepts

At a high level a database is essentially an interface between a user of the system and a collection of data. It is the responsibility of the database system to translate requests from the user to operations that can be executed, and to retrieve and store data. This means that we can identify three key components in a database. First, the syntax used to operate the database, more commonly known as a query language, which will be discussed in more detail in Section 3. Second, some backing store that will be used to store the data in the system, there are many implementations of this database component and we will largely treat it as out of scope for this report, instead viewing it as a black box. Third, a set of concrete operations that a database can execute to bridge the gap between the two aforementioned components, this component will be the main focus of this section.

2.1 Basic Terminology

This section will be used to establish some basic concepts. Most of these concepts are already universally established. However, to avoid any interpretation issues they will be explicitly defined in this section.

2.1.1 Graphs

A graph $\mathcal{G}$ is defined by a set of vertices or nodes $\mathcal{V}$ and a set of edges $\mathcal{E}$. Generally the graphs used in this report will be directed multigraphs. This means that it is allowed for a vertex to have an edge from itself to itself called a self loop and that it is allowed for more than one edge to connect the same two vertices called parallel edges. A graph where these two patterns are not allowed is called a simple graph, directed graphs are sometimes also referred to as digraphs. The opposite of a directed graph is an undirected graph, which simply means that vertices are connected or not, instead of this connection explicitly being from one of the vertices to the other.

2.1.2 Labels

Graphs can generally have two types of labels, vertex labels and edge labels, edge labels are sometimes also referred to as predicates. A label is effectively some extra information attached to either a vertex or an edge and the set of all possible labels $\mathcal{L}$ is called the label set of a graph. For simplicity all of the graphs in this report will only have edge labels, since it is generally trivial to extend operations to take into account vertex labels.

2.1.3 Edges

Edges are the links in a graph that connect vertices. Generally for directed labelled edges we will define an edge e as $(v, u, l) \in \mathcal{V} \times \mathcal{V} \times \mathcal{L}$. This means that an edge is an ordered tuple containing three elements, the source vertex v, the target vertex u, and the edge label l.

2.2 Reachability Queries & Query Output

When querying a graph database, we naturally expect some sort of response back from the database. The exact nature and form of this response can vary in practise, especially since queries can often be used to insert data, retrieve data, and to manage the database. However, in this report we will focus on a specific subset of queries called reachability queries. Reachability queries are one of the most common queries executed in practice and they are also what graph databases excel at compared to relational databases. The goal of a reachability query is to find vertex pairs in the database graph that are connected by a path of edges that satisfies the constraints of the query. The exact nature of these constraints will be discussed further in Section 2.3 and Section 3, but

it is worth noting that this means that the output of a reachability query is a set of pairs of vertices that are connected by a matching path.

Note that by returning only the source and target vertices of these paths we are only implicitly describing the path, and losing any information about the path itself. This is unlike an explicit description of the path, which can be thought of as a sequence of labels that needs to be traversed to go from a specific vertex in the graph to a different vertex. Note that this design choice is essentially analogous to the behaviour of the `where` clause in *SQL*, which also does not provide any detailed information about which expressions evaluated to true to make a row part of the result. Essentially, we only care about the fact that a source-target vertex pair satisfies the query constraints, we do not need to know exactly how a pair satisfied these constraints. Thus, the result of evaluating a query is a set of source-target vertex pairs (s, t) that implicitly represent paths matched by the query in the graph. When the source and target of a path are identical we call the path a loop or cycle.

2.3 Database Operations

In this section we will introduce some concrete query operations that are commonly implemented in databases. It is worth noting that this is not a comprehensive set of all existing database operations, instead we only introduce the operations required to introduce the *RPQ* and *CPQ* query languages in Section 3. Note that operations will be introduced from the perspective of a graph database in this section. However, all of these operations can also be found in relational databases, and their implementation there is often identical or very similar to what will be described in this section. Note that some operations go by multiple names, for clarity all common names are listed in this section, in other parts of this report only a single name will be used.

Finally, each operation section will include a small example showing how the operation is applied to actual input data. It is worth noting that what is shown here is different from the concept of a query graph. A query graph is the visual representation of a query that represents the exact graph sub-structure the query is designed to match, a more complete explanation on how query graphs can be constructed is given in Section 2.2.3 of my thesis [7]. However, note that some queries match multiple sub-structures and therefore cannot be visualised as a single query graph.

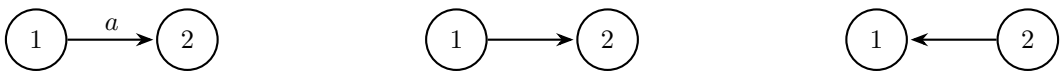
2.3.1 Edge Selection

We start with the most ubiquitous database operation that is used in virtually all queries. The edge or label operation selects all the edges in the database graph with a specific label together with their source and target vertices. Essentially this operation selects all paths of length 1 with a specific label following the direction of the edges. Similarly then, the inverse edge label operation also selects length 1 paths with a specific label. However, this operation traverses edges from target to source. The formal definitions of the edge and inverse edge operations when executed on a database graph $\mathcal{G}$ are given in Equation 1 and 2 respectively with $l \in \mathcal{L}$.

$$[\![l]\!]_{\mathcal{G}} = \{(v, u) \mid (v, u, l) \in \mathcal{E}\} \tag{1}$$

$$[\![l^-]\!]_{\mathcal{G}} = \{(u, v) \mid (v, u, l) \in \mathcal{E}\} \tag{2}$$

A visual representation of forward and inverse edge selection is shown in Figure 1.



(a) The database graph. (b) Edge selection result graph. (c) Inverse edge selection Result graph.

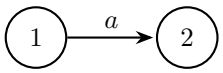
Figure 1: Visual representation of forward and inverse edge selection for label a on a single edge database graph.

2.3.2 Identity

The identity operation simply selects all vertices from the graph with themselves. Essentially, this operation can be thought of as the selection of all zero length paths in the database. The formal definition of this operation when executed on a database graph $\mathcal{G}$ is given in Equation 3.

$$[\![id]\!]_{\mathcal{G}} = \{(v, v) \mid v \in \mathcal{V}\} \tag{3}$$

A visual representation of identity selection is shown in Figure 2.



(a) The database graph.

(b) Identity selection result graph.

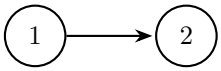
Figure 2: Visual representation of identity selection on a small single edge database graph.

2.3.3 Concatenation or Join

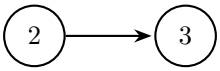
The join or concatenation operator is one of the most frequently used operations, both in graph databases and in relational databases. This operation first evaluates the two sub-queries it is operating on and then joins their result sets, returning pairs where the target vertex of a pair from the first sub-query is equal to the source of a pair from the second sub-query. This process can be thought of as joining two paths that end and start at some shared vertex, these paths are then combined and the source and target of the new combined path are returned. The formal definition of this operation when evaluated on a database graph $\mathcal{G}$ is given in Equation 4, where q_1 and q_2 represent valid queries. Note that this operation is associative, but not commutative.

$$[\![q_1 \circ q_2]\!]_{\mathcal{G}} = \{(v,u) \mid \exists m \in \mathcal{V} : (v,m) \in [\![q_1]\!]_{\mathcal{G}} \wedge (m,u) \in [\![q_2]\!]_{\mathcal{G}}\} \tag{4}$$

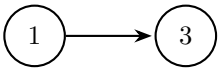
A visual representation of a simple join between two result graphs is shown in Figure 3.



(a) The left input graph.



(b) The right input graph.



(c) The output result graph.

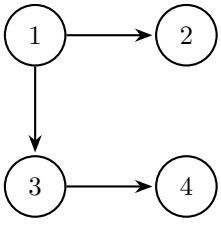
Figure 3: Visual representation of the join operation on two small single edge input graphs.

2.3.4 Intersection or Conjunction

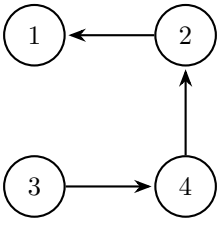
The intersection or conjunction operation is relatively straightforward and can be thought of as the equivalent of the logical `and` operation. This operation first evaluates two sub-queries, and then returns only those pairs that are present in the evaluation result of both evaluated sub-queries. Essentially this process can be seen as running two sub-queries in parallel between the same vertices and only pairs found by both sub-queries are returned. The formal definition of this operation when evaluated on a database graph $\mathcal{G}$ is given in Equation 5, where q_1 and q_2 represent valid queries. Note that this operation is both associative and commutative.

$$[\![q_1 \cap q_2]\!]_{\mathcal{G}} = \{(v,u) \mid (v,u) \in [\![q_1]\!]_{\mathcal{G}} \wedge (v,u) \in [\![q_2]\!]_{\mathcal{G}}\} \tag{5}$$

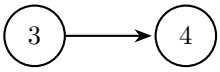
A visual representation of the intersection operation between two result graphs is shown in Figure 4.



(a) The left input graph.



(b) The right input graph.



(c) The output result graph.

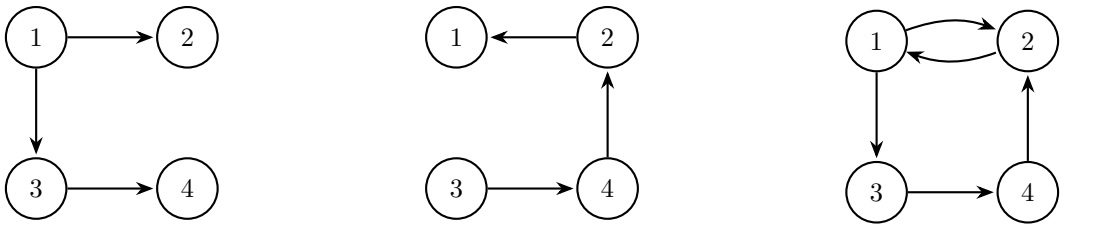
Figure 4: Visual representation of the intersection operation on two input graphs.

2.3.5 Disjunction

The disjunction operation is relatively straightforward and can be thought of as the equivalent of the logical `or` operation. This operation first evaluates two sub-queries, and then returns all the pairs that are present in the evaluation result of either of the two evaluated sub-queries. Essentially we can allow more freedom in the way vertices are connected with this operation. The formal definition of this operation when evaluated on a database graph $\mathcal{G}$ is given in Equation 6, where q_1 and q_2 represent valid queries. Note that this operation is both associative and commutative.

$$[\![q_1 \cup q_2]\!]_{\mathcal{G}} = \{(v,u) \mid (v,u) \in [\![q_1]\!]_{\mathcal{G}} \vee (v,u) \in [\![q_2]\!]_{\mathcal{G}}\} \tag{6}$$

A visual representation of the disjunction operation between two result graphs is shown in Figure 5. It is worth noting that disjunction is one of the operations that cannot be visualised as a query graph as it is not enforcing additional constraints on the query. Instead it is essentially splitting the query into two different queries that need to be evaluated separately, this is also part of the reason why this operation is relatively expensive to evaluate.



(a) The left input graph. (b) The right input graph. (c) The output result graph.

Figure 5: Visual representation of the disjunction operation on two input graphs.

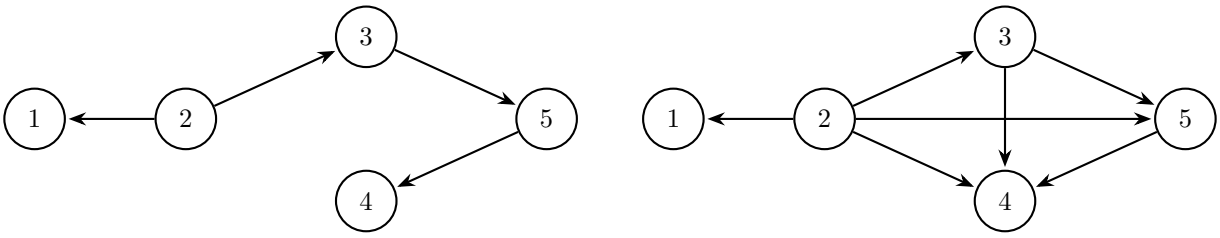
2.3.6 Kleene or Transitive Closure

The Kleene or transitive closure operation is one of the more complex operations and also one of the most expensive operations to execute. It functions similarly to the Kleene star '$*$' found in regular expressions in that it allows a sub-expression to be evaluated repeatedly until the result set no longer grows in size. For regular expressions we often use this to match a string of any length. However, we can also make use of this operation within a graph database, for example, we can use this operation to repeatedly follow edges with a specific label, allowing us to find vertices connected by a path of any length where all edges have some given label. The formal definition of this operation when evaluated on a database graph $\mathcal{G}$ is given in Equation 7, where q represents a valid query.

$$[\![q^*]\!]_{\mathcal{G}} = \{(v,u) \mid (v,u) \in TC([\![q]\!]_{\mathcal{G}})\} \tag{7}$$

Here $TC(R)$ is used to denote the transitive closure of the binary relation R, that is, the smallest binary relation on the set of all distinct source and target vertices in R that both contains R and is transitive. In simpler terms, if a path of any length exists in the input graph between two vertices, then a path of at most length one exists between these vertices in the output graph.

A visual representation of the transitive closure operation between two result graphs is shown in Figure 6. It is worth noting that the transitive closure is one of the operations that cannot be visualised as a query graph due to its recursive nature. This is also why this operation tends to be extremely expensive to evaluate.



(a) The input graph. (b) The output result graph.

Figure 6: Visual representation of the transitive closure operation on an input graph.

3 Query Languages

As mentioned in Section 2, query languages are the primary interface used by users to operate a database. For most *SQL* [2] is probably what comes to mind when thinking of a query language. However, while not incorrect, the concept of a query language is more board. In essence a query language is simply a set of database operations together with a grammar that specifies how these operations are allowed to be combined. Note that this also

means that it is possible for all operations of one query language to be covered by another language, meaning the language is entirely contained within another language. Consequently, this means that the so-called expressive power of query languages is not always equal, and that there may be queries that cannot be answered directly by a specific language[1]. However, these restrictions can enable specific evaluation optimisations that make this trade-off worth while. Finally, it is worth noting that a new ISO standard from 2024 establishes a new query language to serve as the *SQL* for graph databases called *GQL* [1].

3.1 Conjunctive Path Queries (*CPQ*)

As stated in Sasaki et al. [9] and prior research projects [7, 5, 4, 8], conjunctive path queries (*CPQ*) are a basic graph query language. A conjunctive path query can recursively be constructed from the operations of identity 'id', edge label 'l', inverse edge label 'l^-', join '$\circ$' and intersection '$\cap$'. For the edge label operation the labels l come from a finite set of labels $\mathcal{L}$. This results in the grammar shown in Equation 8.

$$CPQ ::= id \mid l \mid l^- \mid CPQ \circ CPQ \mid CPQ \cap CPQ \mid (CPQ) \tag{8}$$

Note that all of these operations have already been covered in more detail in Section 2.3, except for the last operation which simply strips away any outer parenthesis. It is also worth noting that all the operations used in *CPQ* can be visualised as a graphs themselves, meaning that a so-called query graph can be constructed for any *CPQ* [7]. This query graph then represents the exact sub-structure of the result graph paths, meaning the set of all query homomorphic sub-graphs of the database graph is the answer to a *CPQ* [7]. Furthermore, each *CPQ* query graph has a unique core, this means that equivalent queries even if formulated differently will have the same core, which is a useful property for potential optimisations [7, 5].

3.2 Regular Path Queries (*RPQ*)

Regular path queries (*RPQ*) [3] are a basic graph query language that can recursively be constructed from the operations of edge label 'l', inverse edge label 'l^-', join '$\circ$', disjunction '$\cup$', and transitive closure '$*$'. For the edge label operation the labels l come from a finite set of labels $\mathcal{L}$. This results in the grammar shown in Equation 9.

$$RPQ ::= l \mid l^- \mid RPQ \circ RPQ \mid RPQ \cup RPQ \mid RPQ^* \mid (RPQ) \tag{9}$$

Note that all of these operations have already been covered in more detail in Section 2.3, except for the last operation which simply strips away any outer parenthesis.

4 Abstract Query Syntax Tree

Although not really a query language itself, all queries can be converted into an abstract syntax tree (*AST*). An AST is a simple tree data structure where inner nodes have at most two child nodes, that encapsulates the same information as a query that is easier to interpret by a database. In Figure 7 an example AST is shown for the *CPQ* $a \circ (a \cap b)$.

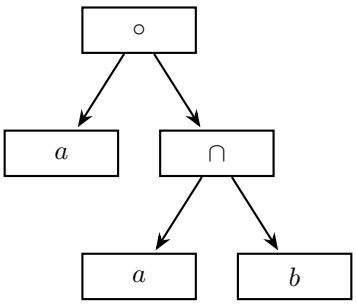


Figure 7: The abstract syntax tree for the *CPQ* $a \circ (a \cap b)$.

[1] Query decomposition can be used to solve this issue

References

[1] ISO/IEC JTC 1. *Information technology — Database languages GQL*. 1st ed. ISO. 2024. 610 pp. URL: https://www.iso.org/standard/76120.html.

[2] ISO/IEC JTC 1. *Information technology — Database languages SQL*. 6th ed. ISO. 2023. 74 pp. URL: https://www.iso.org/standard/76583.html.

[3] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. "Querying graphs". In: *Synthesis Lectures on Data Management* 10.3 (2018), pp. 1–184. URL: https://perso.liris.cnrs.fr/angela.bonifati/pubs/book-Bonifati-et-al-18.pdf.

[4] Roan Hofland. "Conjunctive Path Query Generation for Benchmarking". Capita Selecta Report. Version 2.8. Eindhoven University of Technology, Mar. 21, 2022. 15 pp. URL: https://research.roanh.dev/Conjunctive%20Path%20Query%20Generation%20for%20Benchmarking%20v2.8.pdf.

[5] Roan Hofland. "CPQ Keys: a survey of graph canonization algorithms. Towards a CPQ based graph database index". Internship Report. Version 1.1. Osaka University and Eindhoven University of Technology, July 17, 2022. 30 pp. URL: https://research.roanh.dev/cpqkeys/CPQ%20Keys%20v1.1.pdf.

[6] Roan Hofland. *gMark*. Version 1.2. July 2023. URL: https://github.com/RoanH/gMark.

[7] Roan Hofland. "Indexing Conjunctive Path Queries for Accelerated Query Evaluation". Master's Thesis. Osaka University and Eindhoven University of Technology, Aug. 2, 2023. 94 pp. URL: https://thesis.roanh.dev/.

[8] Roan Hofland. "Subquery Evaluation Using a CPQ-native Index". Seminar Research Proposal. Version 1.3. Eindhoven University of Technology, Jan. 18, 2023. 11 pp.

[9] Yuya Sasaki, George Fletcher, and Makoto Onizuka. "Language-aware indexing for conjunctive path queries". In: *IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE. May 2022, pp. 661–673.

[10] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020. ISBN: 9780078022159. URL: https://www.db-book.com/.