Department of
Mathematics and Computer Science
Artificial Intelligence and Data Engineering Lab

Graduate School of
Information Science and Technology
Onizuka Lab

Indexing Conjunctive Path Queries for Accelerated Query Evaluation

Master's Thesis

by

Roan Hofland

Osaka University, Japan
Eindhoven University of Technology, The Netherlands
roan@roanh.dev

Supervisors:
prof. dr. George Fletcher (Eindhoven)
dr. Yuya Sasaki (Osaka)

Thesis Committee:
prof. dr. George Fletcher (Eindhoven)
dr. Yuya Sasaki (Osaka)
dr. Erik de Vink (Eindhoven)

Osaka, Japan

2nd of August, 2023

Abstract

In the modern world big data is becoming increasingly important and ubiquitous. One prevalent form of data is graph data, which can be used to model various forms of network data, for example social networks. In order to support efficient processing of graph data, graph databases are actively being developed. One general technique to improve the performance of a database is by using an index. The main idea behind an index is to precompute information required to evaluate queries and to store this information such that it can be retrieved efficiently. However, graph databases have not yet been able to enjoy the years of extensive optimisation that traditional relational databases have.

In this thesis we present an approach to develop a graph database index tailored to support the evaluation of conjunctive path queries (*CPQ*). This promising optimisation for graph databases in the form of a *CPQ*-native Index aims to significantly accelerate the evaluation of *CPQ*s, which are one of the most frequently used queries for complex graph analysis [65]. This, combined with the fact that in practice query evaluation times are often important to keep low, makes optimising query evaluation an interesting area of research. The proposed *CPQ*-native Index when deployed can yield significant performance improvements for contemporary businesses and sciences that perform complex graph analyses.

Within this thesis we will first present all the required building blocks to create a *CPQ*-native Index and explain the concept of a database index in more detail. After discussing the individual components, we will then explain in detail how to construct a *CPQ*-native Index and evaluate its performance on real world datasets. Based on this evaluation we will see that at least a limited form of native support for *CPQ*s can be implemented with relative ease in most applications.

Acknowledgements

This thesis project in its current form was only made possible with the help of many people. First and foremost, I would like to thank my two supervisors George Fletcher and Yuya Sasaki for their support not only during my thesis project, but also during my internship and capita selecta project. It was an honour to work together with two passionate researchers for such an extended period of time. I also want to thank Seiji Maekawa for his help during our previous projects and during the start of my thesis. I am also grateful for all the support I have received from my family in going abroad and during my stay abroad.

Due to the special setting of my thesis in Japan, I also owe a thanks to many people for making this logistically possible. The University of Osaka for allowing me to come here, George and Yuya for making all the arrangements surrounding the exchange partnership, various staff members at both universities and the Japanese embassy for arranging all the paper work and my visa, and the staff at Osaka University Global Village Tsukumodai where I stayed during my visit.

Finally, I would like to thank Armin, Mireille, George and Yuya for proofreading and giving feedback on the initial drafts of this thesis.

Roan Hofland
August 2023

Contents

List of Figures

List of Tables

List of Equations

List of Listings

List of Algorithms

1 Introduction 第一章

Nowadays big data plays an integral role in contemporary businesses and sciences. One prevalent form of data is graph data, which is used to model various forms of network data, for example, social networks, road networks, and the topology of the internet. In order to accommodate graph data, we have started to develop so called graph databases specifically tailored towards graph data and the types of tasks often performed with graph data. These tasks are often related to retrieving specific information, finding specific patterns, or deriving certain metrics from data stored in the database and are typically performed by a process called querying. Executing a query is effectively a way to ask a question to a database. Given that in practice queries will often either be executed extremely often or be extremely time-consuming to execute, optimising this querying process is an interesting area of research. This is especially true when it comes to graph databases, as they are still relatively new with fewer already existing optimisations compared to traditional relational databases. Furthermore, one important factor complicating these optimisations is that many components in query evaluation are complex or even intractable for the general case. As such, it is often beneficial to restrict the problem and focus on a subset of all possible input queries, which we can do by defining a query language. The purpose of a query language is to enable optimisations specifically targeting queries that are part of the defined query language, making these queries more efficient to evaluate. Naturally, this approach works best if the majority of queries actually executed are part of the query language targeted for optimisation. A common method to then achieve these optimisations, and the main topic of this thesis, is to make use of an index. The goal of an index is to store precomputed (partial) information about the result of evaluating certain queries. This information can then be retrieved efficiently and combined to form the full result for a query being evaluated. The concept of an index will be explained in more detail in Section 1.1.

The query language we target for optimisation with the index presented in this thesis is the language of conjunctive path queries (*CPQ*). These queries are one of the most frequently used queries for complex graph analyses and were recently formalised in a paper by Sasaki et al. [65]. However, due to their novelty, various domains are still lacking tailored support. One domain where *CPQ* awareness could significantly improve performance is graph database querying, which as mentioned can be achieved using an index. Given this technique and the ubiquity of *CPQ*s, it then makes sense to build an index that can accelerate the evaluation of *CPQ*s, which will be the topic we explore in this thesis.

Within this thesis we aim to present a comprehensive overview of all the components required to build a *CPQ*-native Index, as well a concrete description of how to construct this index. However, while this end goal will always be used to motivate decisions made throughout this thesis, the final index is not the only valuable contribution. Some of the components researched to build the index are valuable research contributions in their own right and some are even more universally applicable. This is especially true for the core and homomorphism related algorithms we will present in Chapter 3. A comprehensive summary of all the research contributions will be given at the end of this thesis in Chapter 6 and a preview will be given in Section 1.3.

Finally, it is worth mentioning that the work presented in this thesis builds on a large body of prior research. We will make use of many fundamental concepts from the literature on graph theory, databases and bisimulation [15, 12, 25, 20, 24, 1]. Perhaps most importantly, we will also heavily reference the paper on language-aware indexing for *CPQ*s by Sasaki et al. [65]. The research groups at Osaka and Eindhoven have also already explored a number of relevant research directions in internal notes [47, 46, 26, 23]. Lastly, I will also be building on my own previous research [39, 35, 34, 33]. All the prior research will be discussed in more detail in Section 2.1.

1.1 What is a Database Index

The goal of an index is essentially to accelerate queries we want to execute on a database. The database can effectively be thought of as a repository of all the knowledge we have available, and the query is the question we want to ask. Naturally, it is always possible to answer a question by going through all the data we have available. However, there are many applications where time is an important factor in answering database queries. Depending on the exact use case it may make sense to use a database index to reduce the time it takes to run certain queries.

The main idea behind an index is to (partially) precompute the result of certain queries. By doing so, the result of one of these queries can simply be returned immediately when it is executed without having to search through all the data. Essentially we are building a large lookup map of queries and their evaluation result. Naturally, there are serious scalability issues involved with the construction of such an index. If we do not know what a user may be interested in asking, we essentially have to build an index storing every possible formulation of every possible question a user may ask together with its result. If we apply this concept to the real world, this effectively results in an infinite number of questions to store, since you can always add more details to an existing question to make a new question. Within a database context, we generally either have artificial limits or hardware limits on the size of queries. However, this does not really solve the underlying issue and generally these limits are as large as possible. Hence, when designing an index we usually have to decide in advance what we will be storing.

To give a real world example. While storing all possible questions and their answer is infeasible, we can scope the question. Naively, you may think that limiting the questions to a specific topic may work here. However, note that this does not address the problem where we can always add more details to an existing question to make a new question. Instead, we have to define more fundamental constraints on a question. The first constraint is to formalise the query language being used. For this thesis we focus on the language of *CPQ*s, in a real world context we could for example decide to limit ourselves to English as defined by the Oxford English dictionary. Doing this gives us a set of axioms to start working from, i.e., labels in the context of a graph database, and words in the context of the English language. The next step is to formalise a set of rules for combining these axioms into larger constructs. For this thesis this will be the formal definition of a *CPQ* as defined in Section 2.2.2, for English language grammar can be seen as a similar concept. Having built this more restrictive framework for asking questions, we can now proceed to define effective limits on these questions.

For example, while scoping to a specific topic does not work as discussed, we can limit questions to a specific number of words. The Oxford English dictionary currently contains 288740 entries, this means that if we limit questions to at most three words we can ask $288740^3 + 288740^2 + 288740 \approx 24$ quadrillion questions. Assuming relatively compact answers, this means that we can store an answer to each of these questions with just a few exabytes of storage. It is also important to note that many of these hypothetical questions are not valid under the grammar rules of the language, significantly reducing the number of answers that need to be stored. The *CPQ* query language makes uses of a similar method to quantify the size of queries called the diameter. We will formally introduce this metric in Section 2.2.2.

Given these analogies we can now give a more concrete answer to the title of this section. A database index is some data structure that stores complete answers to specific (partial) queries called keys in a way that makes them efficient to retrieve. For the *CPQ*-native graph database index that will be presented in this thesis, these keys are the CPQ_k cores that will be defined in Section 2.2.2 and Chapter 3, and the answers are source target pairs that will be formalised in Section 2.2.1. Essentially, an index can be thought of as a simple mapping from keys to precomputed results. The approach comes at the cost of additional storage and precomputation time, while improving the evaluation performance of queries that can leverage the index. A slightly more formal view of an index is that the data in the database is partitioned into blocks by some key, such that the data can be efficiently retrieved using this key.

1.2 Objectives

Ultimately, the main research topic of interest is to investigate if query evaluation can be improved using a *CPQ*-native Index. Unfortunately, this topic is too large for a single thesis to properly explore completely. Therefore, we will focus on two important sub-questions instead and list the remaining parts as future work in Section 6.1. The goal of this thesis is to explore a *CPQ*-native Index that is fully functional, but that does have restrictions on the queries it can be used for to accelerate. In particular, the index we will present in this thesis can only accelerate queries in CPQ_k, which is those *CPQ*s with a diameter of at most k, as we will formalise in Section 2.2.2. The two main objectives in this thesis are:

1) **How can we efficiently compute a CPQ_k core?**
   As mentioned in Section 1.1, the index we will present in this thesis will use CPQ_k cores as keys to look up (partial) evaluation results. Hence, we need to formally establish the exact definition of a *CPQ* core and how one can be computed. Furthermore, we also need to investigate suitable methods to represent this core so it can be stored and compared efficiently. It is also worth noting that many of the components required to compute a core are computationally expensive. The answer to this sub-question will be explored in Chapter 3.

2) **How can we associate CPQ_k cores with blocks in the index?**
   As also mentioned in Section 1.1, our intent is for our constructed index keys to map to blocks that contain (partial) results. Naturally, this means that we need to somehow associate our keys with all the blocks they map to. Currently there is very little existing research on this topic, so this question will require the development of new theory. The new theory that was developed to answer this sub-question will be presented in Chapter 4.

Combining these two main objectives we can phrase the problem statement for this thesis as follows:

Problem Statement: How can we index conjunctive path queries to accelerate query evaluation?

Essentially, within this thesis we will focus on the construction of the index. Many questions related to query evaluation and how to best use the constructed index in a more general setting will remain as future work. As mentioned, some of these future work directions will be listed in Section 6.1.

1.3 Contributions

As the primary goal of this thesis is to develop a *CPQ*-native Index, the completed index is naturally the main contribution. However, this is a composite result supported by many components, some of which are valuable contributions in their own right. On a conceptual level we will introduce many definitions, ideas and constructions required to build the index. The most notable contribution is the design for the completed *CPQ*-native Index itself, but the framework for treating *CPQ* cores as keys is equally important. Many of the ideas we will present will also be accompanied by theoretical results to support them. Most notable here are the formal definitions for query homomorphism, *CPQ* cores, and the theory for computing cores for index blocks. In turn we also present concrete algorithms for many of these theoretical results. On an algorithmic level we will present new algorithms to compute tree decompositions and to compute *CPQ* cores. Finally, a concrete implementation of the entire index and all its individual components is provided as open-source GPL v3.0 licensed [28] repositories[1,2] on GitHub and will be discussed in more detail in Section 2.3.

1.4 Overview

Having introduced the basic ideas and concepts in this chapter, we will give a short overview of the remaining chapters of this thesis. First, in Chapter 2 we will give an overview of all the prior research, main definitions and concepts that will be used throughout the rest of this thesis. Second, in Chapter 3 we will introduce all the main components required to construct the *CPQ*-native Index. Third, in Chapter 4 we will describe the complete *CPQ*-native Index. Fourth, in Chapter 5 we will present an evaluation of the completed index. Finally, we will end the thesis in Chapter 6 by summarising and discussing the main results, as well as presenting a number of possible directions for future work. Here we will also see that we have succeeded in providing an answer to the problem statement in Section 1.2 of how to index *CPQ*s to accelerate query evaluation.

[1] `https://github.com/RoanH/CPQ-native-index`
[2] `https://github.com/RoanH/gMark`

Preliminaries

2 第二章

Before discussing the construction of the *CPQ*-native Index and the components required to build such an index, we will first cover important background knowledge. In this chapter we will start with a detailed overview of the most relevant prior research in Section 2.1. Following this we will introduce all the important terminology and definitions used throughout this thesis in Section 2.2. Finally, in Section 2.3 we will discuss the reference implementation for this thesis in more detail.

2.1 Prior Research

Although predominantly not part of the published literature, the work presented in this thesis does build on a number of important previous research projects. The goal of this section is to briefly cover them all to present a background on the work around *CPQ*s. However, it is worth noting that while the formal definition of *CPQ*s is recent, many closely related topics are not. *CPQ*s are a class of recursively constructed graphs, and classes like this have been extensively studied in the literature for decades [15]. Of particular note are the classes of conjunctive queries (*CQ*) and series-parallel graphs (*SP*). *CPQ*s are a subset of the former and an extension of the latter by the addition of an identity and intersection operator. In addition, it is worth noting that all of the operations used to define the *CPQ* query language have been formally defined before [12, 25]. In fact, the class of graphs itself has also been formalised a number of times before for different use cases [20]. The novelty of the current formalisation as *CPQ* stems from its use as a query language for the acceleration of query evaluation. The definition of a *CPQ* that will be used during this thesis will be provided in Section 2.2.2. The concept of a language-aware index has also been studied before, with the general approach to structural indexing being explained in the "Querying Graphs" book [12]. This book also briefly introduces the notion of bisimulation [24], which we will explain in more detail in Sections 2.1.5 and 2.2.5, and which the language-aware index by Sasaki et al. [65] is based on. In the following sections we will introduce the core resources available that will be used in this thesis that directly cover *CPQ*s.

2.1.1 Language-aware Indexing for Conjunctive Path Queries

First and foremost, the 2022 paper on language-aware indexing for conjunctive path queries by Yuya Sasaki, George Fletcher, and Makoto Onizuka [65]. This paper was published as a collaboration between Osaka University and Eindhoven University of Technology, and served as the original inspiration for this thesis. In this paper language-aware indexing for *CPQ*s is proposed. The proposed method essentially covers an input *CPQ* to evaluate with paths and then uses these paths as keys to the index. This results in an index that is aware of *CPQ*s, but does not natively support them since paths are used as index keys. A variation of the proposed index that allows a user to indicate label sequences of interest is also proposed. This variation results in a more scalable index at the cost of only being able to accelerate the processing of queries with the indicated label sequences. We refer to an index where the user is able to specify what the index should store as interest-aware and this is an extremely powerful technique to mitigate the storage issues raised in Section 1.1.

A natural next research step, and the topic of this thesis, would be to natively integrate *CPQ*s in the design of the index such that they can directly be used as index keys. Also note that the index proposed in the language-aware indexing paper already partitions all paths in the graph into partition blocks that are indistinguishable with respect to the query language using the notion of k-path-bisimulation. This is a major component of the index which we will largely reuse for the *CPQ*-native Index described in Section 4. The reference implementation for the language-aware index is available on GitHub [63] under the MIT license [55].

2.1.2 Internal Notes

Given that some form of *CPQ*s are a topic that has been studied for a while at both the Eindhoven University of Technology in the Netherlands and Osaka University in Japan, there are a number of unpublished internal resources available [47, 46, 26, 23, 39]. While the language-aware indexing paper [65] largely covers the the main topics, some specific aspects have already been explored in greater detail.

Most relevant to this thesis are unpublished notes on designing a *CPQ*-Core-based Index written by Seiji Maekawa from Osaka University [47, 46]. These notes describe how to compute the query graph of a *CPQ* (see Section 2.2.3) and provide a nearly complete theory on computing *CPQ* cores (see Section 3.2) based on older notes [26]. Some notes on designing a *CPQ*-native Index, as well as simple bounds on the total number of cores are also presented. This document of internal notes can effectively be seen as the initial foundation for this thesis, with all the concepts introduced in these notes being expanded upon in this thesis. My own seminar research proposal [39] also provides a basis for this thesis.

2.1.3 *CPQ* Keys

During an internship at the University of Osaka, I conducted a survey of graph canonisation algorithms [35, 34]. As mentioned in Section 1.1, the goal of the thesis project is to use *CPQ*s as index keys. In order to be able to do this we need a method to turn a *CPQ*, which is in essence a graph, into a canonical form that can be used as a key. For example, a string representation of the graph could be used as long as the string is equal for all isomorphically equivalent graphs. The empirical study conducted during my internship evaluating graph canonisation algorithms on their performance on *CPQ* query graphs, showed that the sparse version of nauty [54] is most suited for this purpose. We will make use of this result when discussing our approach to *CPQ* canonisation in Section 3.4 and we will also reuse a number of the algorithms implemented during this project.

2.1.4 gMark

During a Capita Selecta project I worked on random *CPQ* generation for benchmarking purposes [33]. The developed algorithm was heavily inspired by gMark [6], which already implemented similar query generation for the regular path query (*RPQ*) query language. During this project I started a rewrite of the gMark software and this codebase was also further extended with more *CPQ* related utility features during my internship [35]. Currently, this version of gMark is essentially a piece of software focussed primarily around the development of *CPQ* related utilities and comes with an extensive API to make it easy to test new ideas involving *CPQ*s. For this thesis gMark was again extended further with more general utilities, as we will discuss in more detail in Section 2.3.1.

2.1.5 Bisimulation

While we are primarily interested in the notion of k-path-bisimulation in this thesis, the general field of bisimilarity is much larger. The notion of bisimulation has been studied extensively throughout the literature [1]. Most notably, efficient algorithms for finite input that work by refining partition blocks similar to the approach for k-path-bisimulation, have been researched for decades. These classical algorithms for computing bisimilarty generally have a runtime that varies in the number of states n and transitions m in the input, such as $\mathcal{O}(n \cdot m)$ [43] or $\mathcal{O}(m \log n)$ [57]. Note that for a graph states and transitions correspond to vertices and edges respectively. It is also worth mentioning that the algorithm used by the language-aware index has polynomial time complexity [65]. We will discuss the notion of k-path-bisimulation which we will work with in more detail in Section 2.2.5.

2.2 Terminology

The main content of this thesis relies on a number of important pre-existing definitions and concepts. These topics will be introduced in this section and will form the basis for all new theory introduced in the later chapters of this thesis. At the same time this section doubles as an easy to reference overview of all the assumed prior knowledge, definitions and conventions.

2.2.1 Basic Terminology

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

Graphs

Naturally graphs play an important role in a thesis about graph database query evaluation. 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 in this thesis 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.

Labels

Graphs can generally have two types of labels, vertex labels and edge labels. 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. Most of the graphs in this thesis will only have edge labels.

Edges

Since we will be dealing with different types of graphs in this thesis, we also have more than one definition of an edge. Generally for labelled edges we will define an edge e as $(v, u, l) \in \mathcal{V} \times \mathcal{V} \times \mathcal{L}$. Similarly, if no edge labels are involved we will use $(v, u) \in \mathcal{V} \times \mathcal{V}$. If the graph is directed all edges will be from v to u, otherwise the order is irrelevant.

Tree

A common special type of graph used in this thesis is a tree. A tree is a directed graph where all edges are directed away from a special vertex called the root. Furthermore, a tree is not allowed to have any cycles, meaning that when traversing a tree you will never visit a specific node or edge more than once unless you follow an edge more than once.

Bipartite Graph

Another special type of graph is a bipartite graph. In a bipartite graph the vertex set $\mathcal{V}$ can be split into two subsets $\mathcal{V}_1, \mathcal{V}_2 \subset \mathcal{V}$ that do not share any elements $\mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset$, such that all edges are between vertices in $\mathcal{V}_1$ and $\mathcal{V}_2$. This means that there are no edges connecting vertices within the same subset.

Sets

Sets are commonly presented as unordered constructs. However, within this thesis, notably in Section 3.1.3, it is beneficial to think of sets as being ordered. Since sets being ordered does not affect other parts of this thesis we will by default assume that a set is ordered. Consequently, any loops over a set will return set elements in order.

Paths

In this thesis we will often refer to paths that exist in a graph either explicitly or implicitly. An explicit reference to a path 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. Intuitively, the length of this path is then the number of labelled edges that has to be traversed. However, more often than not we will not care about the exact path in this thesis. In such contexts we will instead implicitly refer to the existence of paths by simply giving a so called source target pair. Such a pair consists of two vertices called the source and target that are connected by some path, but we do

not further specify exactly how this path looks. This concept further ties into the definition of the source and target vertex in a query, as will be explained in more detail in Section 2.2.2 and 2.2.3. When the source and target of a path are identical we call the path a loop or cycle. Sometimes we will refer to the set of all paths in a graph of at most a given length k, we will denote this with $\mathcal{P}^{\leq k}$.

2.2.2 Conjunctive Path Queries

As stated in Sasaki et al. [65] and prior research projects [35, 33, 39], 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 or concatenation '$\circ$' and conjunction or 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 2.1.

$$CPQ ::= id \mid l \mid l^- \mid CPQ \circ CPQ \mid CPQ \cap CPQ \mid (CPQ) \tag{2.1}$$

The exact definition of these operations when evaluated on a directed multigraph $\mathcal{G}$ with vertex set $\mathcal{V}$, edge label set $\mathcal{L}$ and edge set $\mathcal{E} = \mathcal{V} \times \mathcal{V} \times \mathcal{L}$ will be discussed next based on Equations 2.2, 2.3, 2.4, 2.5, 2.6 and 2.7.

$$[\![id]\!]_{\mathcal{G}} = \{(v, v) \mid v \in \mathcal{V}\} \tag{2.2}$$
$$[\![l]\!]_{\mathcal{G}} = \{(v, u) \mid (v, u, l) \in \mathcal{E}\} \tag{2.3}$$
$$[\![l^-]\!]_{\mathcal{G}} = \{(u, v) \mid (v, u, l) \in \mathcal{E}\} \tag{2.4}$$
$$[\![q_1 \circ q_2]\!]_{\mathcal{G}} = \{(v, u) \mid \exists m \in \mathcal{V} : (v, m) \in [\![q_1]\!]_{\mathcal{G}} \wedge (m, u) \in [\![q_2]\!]_{\mathcal{G}}\} \tag{2.5}$$
$$[\![q_1 \cap q_2]\!]_{\mathcal{G}} = \{(v, u) \mid (v, u) \in [\![q_1]\!]_{\mathcal{G}} \wedge (v, u) \in [\![q_2]\!]_{\mathcal{G}}\} \tag{2.6}$$
$$[\![(q_1)]\!]_{\mathcal{G}} = [\![q_1]\!]_{\mathcal{G}} \tag{2.7}$$

These equations show that the result of evaluating a *CPQ* is a set of vertex pairs. As discussed in Section 2.2.1 these pairs should be thought of as paths between the first vertex and the second vertex. Next we will discuss each of these operations in greater detail. Starting with identity in Equation 2.2, this 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. Moving to the edge label operation in Equation 2.3, this operation selects the source vertex of every edge with label l together with the target vertex. Essentially this operation selects all paths of length 1 with a specific label following direction of the edges. Similarly then the inverse edge label operation in Equation 2.4 also selects length 1 paths with a specific label. However, this operation traverses edges from target to source. The join operation in Equation 2.5 evaluates the two *CPQ*s it is joining and then returns pairs where the target vertex of a pair from the first *CPQ* is equal to the source of a pair from the second *CPQ*. 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. Next is the conjunction operation from Equation 2.6, this operation again evaluates two *CPQ*s. However, this time the returned pairs are simply those pairs that are present in the evaluation result of both evaluated *CPQ*s. Essentially this process can be seen as running two *CPQ*s in parallel between the same vertices and only pairs found by both *CPQ*s are returned. Finally, the bracket operation from Equation 2.7 is trivial and simply removes any round brackets surrounding a *CPQ* to evaluate.

As previously stated, this means that the result of evaluating a *CPQ* is a set of vertex pairs representing paths that exist within the graph. For convenience it is possible to omit the explicit definition of inverse edge label 'l^-' and to instead extend the label set $\mathcal{L}$ with all inverse labels. However, we will not do that in this thesis since there are a number of places where the distinction between a forward label and an inverse label is important. Nevertheless, for some applications this simplification may be justified. For a better intuition of the patterns *CPQ*s match more visual examples of *CPQ*s will be given in Section 2.2.3.

During index development later on, it is important that we can somehow quantify the size of a *CPQ*. For this purpose we will define the concept of the diameter of a *CPQ* as the maximum number of edge labels to which the join operation is applied. Essentially this can be thought of as the length of the longest path from source to target within the *CPQ* itself. Given the recursive nature of *CPQ*s, we also give a recurrence to compute the diameter of a *CPQ* in Equations 2.8, 2.9, 2.10, 2.11, 2.12 and 2.13.

$$\text{dia}(id) = 0 \tag{2.8}$$
$$\text{dia}(l) = 1 \tag{2.9}$$
$$\text{dia}(l^-) = 1 \tag{2.10}$$
$$\text{dia}(q_1 \circ q_2) = \text{dia}(q_1) + \text{dia}(q_2) \tag{2.11}$$
$$\text{dia}(q_1 \cap q_2) = \max(\text{dia}(q_1), \text{dia}(q_2)) \tag{2.12}$$
$$\text{dia}((q_1)) = \text{dia}(q_1) \tag{2.13}$$

Recalling the intuition of the individual *CPQ* operations given earlier, these definitions are fairly straightforward. As mentioned identity is effectively selecting zero length paths and thus as shown in Equation 2.8 the diameter of identity is zero. Similarly, since forward label and inverse label select length one paths, their diameter is also one as shown in Equations 2.9 and 2.10. For the diameter of the join operation in Equation 2.11 note that this operation effectively concatenates two paths. Therefore, the length of the resulting path is simply the combined length of the paths it was created from. Note that the conjunction operation is intersecting two paths, if we are interested in the longest path then we simply need to look at the length of the longest path in the intersection as shown in Equation 2.12. Finally, the bracket operation shown in Equation 2.13 is again trivial and simply defers the computation to a different rule in the recursion.

When developing the index, we will frequently need to refer to a subset of all *CPQ*s with at most a given diameter k, we will denote this with CPQ_k. More information about *CPQ*s and their properties can be found in the paper by Sasaki et al. [65]. To illustrate how *CPQ*s work, we will show an example of how to construct a query to find people who know someone who knows them. Note that finding people you know who also know you is equivalent to finding a path consisting of two knows edges that starts and ends at the same node. We can write the following *CPQ* for this $(\texttt{knows} \circ \texttt{knows}) \cap id$, where we add the conjunction with identity to ensure we only return results that start and end at the same node. To make the example more concrete we show a simple social network in Figure 2.1. Evaluating the suggested *CPQ* then gives the following results $[\![(\texttt{knows} \circ \texttt{knows}) \cap id]\!]_{\mathcal{G}} = \{(\text{Alice}, \text{Alice}), (\text{Bob}, \text{Bob})\}$.

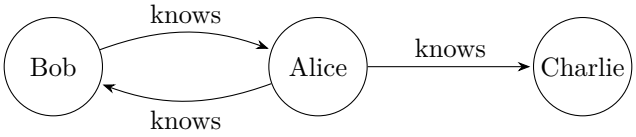


Figure 2.1: A simple social network graph $\mathcal{G}$.

2.2.3 *CPQ* Query Graphs

Since *CPQ* is a graph querying language, we can visualise the graph structure matched by a *CPQ* as a graph. We call this graph the query graph of a *CPQ*. A query graph for a *CPQ* q is defined as $\mathcal{G}_q = (\mathcal{V}_q, \mathcal{E}_q, \mathcal{L}, v_s, v_t)$. Here $\mathcal{V}_q$ and $\mathcal{E}_q$ respectively represent the vertices and the edges in the query graph. The set $\mathcal{L}$ represents the edge labels appearing in the *CPQ*, and v_s and v_t are the source and target vertex of the *CPQ*. Next we will show how to recursively construct the query graph of a *CPQ* using a recursive set of equations. This construction process is identical to the one given in my internship report [35] and based on notes by Seiji Maekawa [47].

Before discussing the construction of a query graph, we first extend its definition with an extra set $\mathcal{F}_{id}$ that keeps track of distinct vertices that should eventually be merged to become the same vertex. This set is required to properly handle the identity operation. The query graph construction works in three main steps. First we have three base case definitions of $f_{q2graph}$ that define a query graph for identity, edge label and inverse edge label. Second we have three recursive definitions for the join, conjunction and bracket operations that compute the query graph of smaller parts of the input query and then merge the results. Finally, we have a recursive merge step f_{merge} that cleans up the final graph by merging marked pairs of identity vertices in $\mathcal{F}_{id}$. After all identity pairs have been merged the set $\mathcal{F}_{id}$ is empty and no longer required. Before discussing the three construction stages in detail, we first clarify the general definition of the query graph construction formula. Note that in general a call to $f_{q2graph}$ takes the form shown in Equation 2.14.

$$f_{q2graph}(q, v, u, \mathcal{G}_q) = \mathcal{G}'_q \tag{2.14}$$

Here q is the *CPQ* to compute the query graph of and $\mathcal{G}_q$ the query graph computed so far, the returned graph $\mathcal{G}'_q$ is then the updated query graph with the parts for q. The vertices v and u are the source and target vertices of the part of the query graph being processed, for the initial call to $f_{q2graph}$ these vertices will be set to the source and target of the entire query graph v_s and v_t respectively. Finally, it is important to note in the initial call that the source and target vertex are already part of the query graph being constructed. Putting everything together this means that the initial call to $f_{q2graph}$ takes the following arguments for a *CPQ* q to convert $f_{q2graph}(q, v_s, v_t, (\{v_s, v_t\}, \emptyset, \mathcal{L}, v_s, v_t, \emptyset))$. Thus, initially $\mathcal{V}_q = \{v_s, v_t\}$, $\mathcal{E}_q = \emptyset$, $\mathcal{F}_{id} = \emptyset$ and the label set $\mathcal{L}$ is determined entirely in advance. Next we will discuss all three stages of the query graph construction process in more detail.

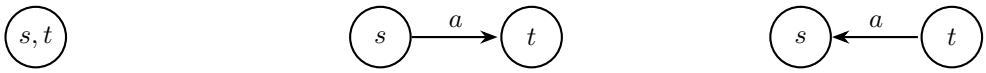
Starting with the base cases of the $f_{q2graph}$ construction. Equations 2.15, 2.16 and 2.17 show the construction method for the identity, forward edge label and inverse edge label operations respectively.

$$f_{q2graph}(id, v, u, (\mathcal{V}_f, \mathcal{E}_f, \mathcal{L}, v_s, v_t, \mathcal{F}_{id})) = (\mathcal{V}_f, \mathcal{E}_f, \mathcal{L}, v_s, v_t, \mathcal{F}_{id} \cup \{(v, u)\}) \tag{2.15}$$

$$f_{q2graph}(l, v, u, (\mathcal{V}_f, \mathcal{E}_f, \mathcal{L}, v_s, v_t, \mathcal{F}_{id})) = (\mathcal{V}_f, \mathcal{E}_f \cup \{v, u, l\}, \mathcal{L}, v_s, v_t, \mathcal{F}_{id}) \tag{2.16}$$

$$f_{q2graph}(l^-, v, u, (\mathcal{V}_f, \mathcal{E}_f, \mathcal{L}, v_s, v_t, \mathcal{F}_{id})) = (\mathcal{V}_f, \mathcal{E}_f \cup \{u, v, l\}, \mathcal{L}, v_s, v_t, \mathcal{F}_{id}) \tag{2.17}$$

To further illustrate the behaviour of these equations we also show their visual output in Figure 2.2 when evaluated with initial call arguments and with $\mathcal{L} = \{a\}$. Here Figure 2.2b shows the forward label construction for $q = a$ and Figure 2.2c the inverse label construction for $q = a^-$. Given that these base cases handle paths of length one as described in Section 2.2.2, their behaviour is relatively straightforward. However, the identity construction does not add any vertices or edges to the graph, it only saves the pair (v, u) to $\mathcal{F}_{id}$. This pair is essentially a note for the f_{merge} construction step we will cover later, saying that while currently separate vertices, v and u are supposed to be the same vertex. Note that intuitively we would expect the query of just identity $q = id$ to look as shown in Figure 2.2a.



(a) Intuition for the query $q = id$. (b) Graph for $f_{q2graph}(a, v_s, v_t, \mathcal{G}_q)$. (c) Graph for $f_{q2graph}(a^-, v_s, v_t, \mathcal{G}_q)$.

Figure 2.2: *CPQ* query graph construction base cases.

Next we will discuss the recursive cases of the $f_{q2graph}$ construction. Again we will start by giving the formal definition for the construction of the join, conjunction and bracket operations in Equation 2.18, 2.19 and 2.20 respectively. Note that two of these formulas feature a special $\cup_G$ operator. This operator is shorthand for taking the union of two query graphs, which we define as follows $(\mathcal{V}_1, \mathcal{E}_1, \mathcal{L}, v_s, v_t, \mathcal{F}_1) \cup_{\mathcal{G}} (\mathcal{V}_2, \mathcal{E}_2, \mathcal{L}, v_s, v_t, \mathcal{F}_2) = (\mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{E}_1 \cup \mathcal{E}_2, \mathcal{L}, v_s, v_t, \mathcal{F}_1 \cup \mathcal{F}_2)$. Essentially the union of two query graphs is simply the union of all the individual sets that make up each query graph, with only $\mathcal{L}$ being an exception to this rule.

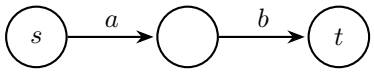
$$\begin{aligned} f_{q2graph}(q_1 \circ q_2, v, u, (\mathcal{V}_f, \mathcal{E}_f, \mathcal{L}, v_s, v_t, \mathcal{F}_{id})) = {} & f_{q2graph}(q_1, v, m, (\mathcal{V}_f \cup \{m\}, \mathcal{E}_f, \mathcal{L}, v_s, v_t, \mathcal{F}_{id})) \\ & \cup_{\mathcal{G}} f_{q2graph}(q_2, m, u, (\mathcal{V}_f \cup \{m\}, \mathcal{E}_f, \mathcal{L}, v_s, v_t, \mathcal{F}_{id})) \end{aligned} \tag{2.18}$$

$$\begin{aligned} f_{q2graph}(q_1 \cap q_2, v, u, (\mathcal{V}_f, \mathcal{E}_f, \mathcal{L}, v_s, v_t, \mathcal{F}_{id})) = {} & f_{q2graph}(q_1, v, u, (\mathcal{V}_f, \mathcal{E}_f, \mathcal{L}, v_s, v_t, \mathcal{F}_{id})) \\ & \cup_{\mathcal{G}} f_{q2graph}(q_2, v, u, (\mathcal{V}_f, \mathcal{E}_f, \mathcal{L}, v_s, v_t, \mathcal{F}_{id})) \end{aligned} \tag{2.19}$$

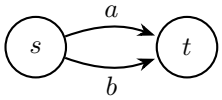
$$f_{q2graph}((q_1), v, u, (\mathcal{V}_f, \mathcal{E}_f, \mathcal{L}, v_s, v_t, \mathcal{F}_{id})) = f_{q2graph}(q_1, v, u, (\mathcal{V}_f, \mathcal{E}_f, \mathcal{L}, v_s, v_t, \mathcal{F}_{id})) \tag{2.20}$$

Similar to the base cases we also show the visual output of the recursive cases in Figure 2.3 when evaluated with the initial call arguments and with $\mathcal{L} = \{a, b\}$. However, the recursive case for brackets is excluded as it has no visual significance and effectively just functions as a pass through to unpack the *CPQ* inside the brackets. For the other two equations, the join case is shown in Figure 2.3a for $q = a \circ b$, note from Section 2.2.2 that intuitively the join operator connects two existing paths into a new longer path. For q note that both $f_{q2graph}(a, v_s, v_t, \mathcal{G}_q)$ and $f_{q2graph}(b, v_s, v_t, \mathcal{G}_q)$ will be computed, producing two subgraphs equal to Figure 2.2b except for the label. These two paths of length one are then combined into the final path as seen in Figure 2.3a. It is also worth noting that the join operation is the only operation that increases the size of the query graph vertex set. The conjunction case is shown in Figure 2.3b for $q = a \cap b$, recall from Section 2.2.2 that conjunction

can be seen as running two *CPQ*s in parallel between the same vertices. Note that in this case these *CPQ*s were again formed by $f_{q2graph}(a, v_s, v_t, \mathcal{G}_q)$ and $f_{q2graph}(b, v_s, v_t, \mathcal{G}_q)$ just like for the join example, and observe that the result in Figure 2.3b has exactly these two length one paths in parallel between the same vertices.



(a) Graph for $f_{q2graph}(a \circ b, v_s, v_t, \mathcal{G}_q)$.



(b) Graph for $f_{q2graph}(a \cap b, v_s, v_t, \mathcal{G}_q)$.

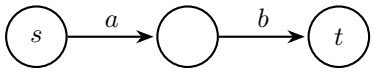
Figure 2.3: *CPQ* query graph construction recursive cases.

Having covered the base cases and the recursive cases we can now cover the final part of the query graph construction that deals with the identity operation. As noted earlier, the set $\mathcal{F}_{id}$ contains pairs of vertices that are supposed to be the same vertex, but that currently are separate vertices. To finish the query graph construction this means that we need to merge these vertices into the same vertex, for which we will introduce a final recursive f_{merge} function shown in Equation 2.21 and 2.22.

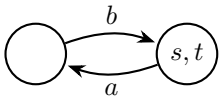
$$f_{merge}((\mathcal{V}_f, \mathcal{E}_f, \mathcal{L}, v_s, v_t, \mathcal{F}_{id})) \text{ if } (\mathcal{F}_{id} \neq \emptyset) = \begin{cases} \{\text{Let } (v_{id}, u_{id}) \text{ be an element in } \mathcal{F}_{id}\} \\ f_{merge}(\\ \quad \mathcal{V}_f \setminus \{v_{id}\}, \\ \quad \mathcal{E}_f \cup \{(u_{id}, u, l) \mid (v, u, l) \in \mathcal{E}_f \wedge v = v_{id}\} \\ \qquad \cup \{(v, u_{id}, l) \mid (v, u, l) \in \mathcal{E}_f \wedge u = v_{id}\} \\ \qquad \cup \{(u_{id}, u_{id}, l) \mid (v, u, l) \in \mathcal{E}_f \wedge v = v_{id} \wedge u = v_{id}\} \\ \qquad \setminus \{(v, u, l) \mid (v, u, l) \in \mathcal{E}_f \wedge (v = v_{id} \vee u = v_{id})\}, \\ \quad \mathcal{L}, \\ \quad v_s \text{ if } v_s \neq v_{id} \text{ otherwise } u_{id}, \\ \quad v_t \text{ if } v_t \neq v_{id} \text{ otherwise } u_{id}, \\ \quad \mathcal{F}_{id} \cup \{(u_{id}, u) \mid (v, u) \in \mathcal{F}_{id} \wedge v = v_{id}\} \\ \qquad \cup \{(v, u_{id}) \mid (v, u) \in \mathcal{F}_{id} \wedge u = v_{id}\} \\ \qquad \setminus \{(v, u) \mid (v, u) \in \mathcal{F}_{id} \wedge (v = v_{id} \vee u = v_{id})\} \\ ) \end{cases} \tag{2.21}$$

$$f_{merge}((\mathcal{V}_f, \mathcal{E}_f, \mathcal{L}, v_s, v_t, \mathcal{F}_{id})) \text{ if } (\mathcal{F}_{id} = \emptyset) = (\mathcal{V}_f, \mathcal{E}_f, \mathcal{L}, v_s, v_t) \tag{2.22}$$

This function consists of a base case shown in Equation 2.22 for when there are no pairs left in $\mathcal{F}_{id}$ and a recursive case shown in Equation 2.21 that consumes one pair from $\mathcal{F}_{id}$ for each call. To illustrate the functionality of the merge step we give a simple query graph for $q = (a \circ b) \cap id$ in Figure 2.4a, for which the merge step has not been executed yet. Note that at this point the set $\mathcal{F}_{id}$ will contain the pair (s, t). After merging s and t into the same vertex using f_{merge} we obtain the final query graph shown in Figure 2.4b.



(a) Graph for $(a \circ b) \cap id$ before the merge step.



(b) Graph for $(a \circ b) \cap id$ after the merge step.

Figure 2.4: *CPQ* query graph construction merge step example.

Having explained all the individual steps, we now give one final complete example for the query $q = (a \cap id) \circ (a \cap b)$. Note that in order to compute the query graph of this *CPQ*, we need to evaluate $f_{merge}(f_{q2graph}((a \cap id) \circ (a \cap b), v_s, v_t, (\{v_s, v_t\}, \emptyset, \{a, b\}, v_s, v_t, \emptyset)))$. Recall that initially the vertex set $\mathcal{V}$ contains only the source vertex v_s and target vertex v_t, while the edge set $\mathcal{E}$ and $\mathcal{F}_{id}$ set are empty. Note that the label set $\mathcal{L}$ is not changed at all by the query graph construction and instead contains all labels that appear in q from the start, giving it the initial value of $\{a, b\}$. For the evaluation we will start with evaluating $f_{q2graph}$ to obtain the input for f_{merge}, this evaluation is shown in Equations 2.23, 2.24, 2.25, 2.26, 2.27 and 2.28. Note that all the discussed variants of $f_{q2graph}$ are present in these steps except for inverse edge label.

$$f_{q2graph}((a \cap id) \circ (a \cap b), v_s, v_t, (\{v_s, v_t\}, \emptyset, \{a, b\}, v_s, v_t, \emptyset)) = \tag{2.23}$$

$$f_{q2graph}((a \cap id), v_s, m, (\{v_s, v_t, m\}, \emptyset, \{a, b\}, v_s, v_t, \emptyset)) \\ \cup_{\mathcal{G}} f_{q2graph}((a \cap b), m, v_t, (\{v_s, v_t, m\}, \emptyset, \{a, b\}, v_s, v_t, \emptyset)) = \tag{2.24}$$

$$f_{q2graph}(a \cap id, v_s, m, (\{v_s, v_t, m\}, \emptyset, \{a, b\}, v_s, v_t, \emptyset)) \\ \cup_{\mathcal{G}} f_{q2graph}(a \cap b, m, v_t, (\{v_s, v_t, m\}, \emptyset, \{a, b\}, v_s, v_t, \emptyset)) = \tag{2.25}$$

$$f_{q2graph}(a, v_s, m, (\{v_s, v_t, m\}, \emptyset, \{a, b\}, v_s, v_t, \emptyset)) \\ \cup_{\mathcal{G}} f_{q2graph}(id, v_s, m, (\{v_s, v_t, m\}, \emptyset, \{a, b\}, v_s, v_t, \emptyset)) \\ \cup_{\mathcal{G}} f_{q2graph}(a, m, v_t, (\{v_s, v_t, m\}, \emptyset, \{a, b\}, v_s, v_t, \emptyset)) \\ \cup_{\mathcal{G}} f_{q2graph}(b, m, v_t, (\{v_s, v_t, m\}, \emptyset, \{a, b\}, v_s, v_t, \emptyset)) = \tag{2.26}$$

$$(\{v_s, v_t, m\}, \{(v_s, m, a)\}, \{a, b\}, v_s, v_t, \emptyset) \\ \cup_{\mathcal{G}} (\{v_s, v_t, m\}, \emptyset, \{a, b\}, v_s, v_t, \{(v_s, m)\}) \\ \cup_{\mathcal{G}} (\{v_s, v_t, m\}, \{(m, v_t, a)\}, \{a, b\}, v_s, v_t, \emptyset) \\ \cup_{\mathcal{G}} (\{v_s, v_t, m\}, \{(m, v_t, b)\}, \{a, b\}, v_s, v_t, \emptyset) = \tag{2.27}$$

$$(\{v_s, v_t, m\}, \{(v_s, m, a), (m, v_t, a), (m, v_t, b)\}, \{a, b\}, v_s, v_t, \{(v_s, m)\}) \tag{2.28}$$

More specifically, from Equation 2.23 to 2.24 we see the join operation being evaluated using Equation 2.18. Then from Equation 2.24 to 2.25 the surrounding brackets are removed using Equation 2.20. This in turn results in two evaluations of the conjunction operation in Equation 2.6 from Equation 2.25 to 2.26. After this we end up with four base case conditions for edge label and identity, which are evaluated from Equation 2.26 to 2.27 using Equations 2.16 and 2.15. Finally, we obtain the output in Equation 2.28.

The next step is to evaluate f_{merge} using the output from $f_{q2graph}$ obtained in Equation 2.28. This process is shown in Equations 2.29, 2.30 and 2.31.

$$f_{merge}((\{v_s, v_t, m\}, \{(v_s, m, a), (m, v_t, a), (m, v_t, b)\}, \{a, b\}, v_s, v_t, \{(v_s, m)\})) = \tag{2.29}$$

$$f_{merge}((\{v_t, m\}, \{(m, m, a), (m, v_t, a), (m, v_t, b)\}, \{a, b\}, m, v_t, \emptyset)) = \tag{2.30}$$

$$(\{v_t, m\}, \{(m, m, a), (m, v_t, a), (m, v_t, b)\}, \{a, b\}, m, v_t) \tag{2.31}$$

In these equations we see that first the version of f_{merge} is used that takes a pair from $\mathcal{F}_{id}$ in Equation 2.29 to 2.30. Processing this pair (v_s, m) results in renaming all occurrences of v_s with m and removing v_s from the graph. Note that the identity of the source vertex is changed from v_s to m during this process. After this step the other definition of f_{merge} is used from Equation 2.30 to 2.31, this definition simply removes the empty set $\mathcal{F}_{id}$ from the query graph data and outputs the final result. The final result is given in Equation 2.31 and represents a query graph with two vertices and three edges, a visual representation of this query graph is given in Figure 2.5. Note that this figure shows the names used for vertices during the construction process, a fully constructed query graph has no vertex labels. However, also note that the vertex labelled m is designated as the source node and the vertex labelled v_t is designated as the target node.

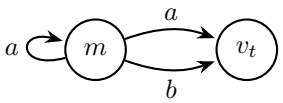


Figure 2.5: Final output graph of the example query graph construction for $(a \cap id) \circ (a \cap b)$.

2.2.4 Language-aware Index

In contrast to a language-unaware index, a language-aware index is designed with knowledge about the graph query language that will be used to query it [65, 12, 47]. Information about the language can then be used to optimise certain aspects of the query evaluation process. Most notably, as the expressiveness of most query languages is limited, there will be paths that cannot be distinguished by any query in a query language. This property makes it possible to process paths that cannot be distinguished by the query language together instead of individually. More formally, given a query language L the goal is to partition the set of all paths in a graph $\mathcal{G}$ into L-equivalence classes also called blocks. The index will then be constructed from these equivalence

classes instead of dealing directly with the set of all paths in the graph. Paths that are assigned to the same L-equivalence class cannot be distinguished by any query in L. In other words, for a query $q \in L$, either all paths from an L-equivalence class appear in $[\![q]\!]_{\mathcal{G}}$ or none appear. Therefore, we can retrieve matching L-equivalence classes for a given input query instead of querying all paths directly to achieve the joint processing of undistinguishable paths. Note that the limited expressiveness of most query languages is by design. Many graph problems related to query evaluation are intractable, thus limiting the expressiveness of languages used for querying introduces new opportunities for optimisation, for example via bounded treewidth as we will discuss in Section 2.2.8.

To conclude we will give a small example to illustrate how paths can be undistinguishable with respect to a query language. For this example assume that we use a query language that can only be used to query a simple path given as a sequence of edge labels. Essentially this language will consist of just the join and label operations as defined for the *CPQ* query language in Section 2.2.2. Now suppose we want to run a query on the graph $\mathcal{G}_{ex}$ given in Figure 2.6 that returns only the pair $(1, 4)$. Observe that this is not possible using a language that only matches simple paths as the path from 1 to 4 and the path from 1 to 5 cannot be distinguished. For example, evaluating the query $a \circ b$ will return $\{(1, 4), (1, 5)\}$, while evaluating $a \circ b \circ c$ will return $\{(1, 6)\}$. This phenomenon is what we refer to when stating that paths are indistinguishable with respect to a certain query language. Incidentally, note that *CPQ* can distinguish these paths, for example: $[\![a \circ b \circ (id \cap (c \circ c^-))]\!]_{\mathcal{G}_{ex}} = \{(1, 4)\}$.

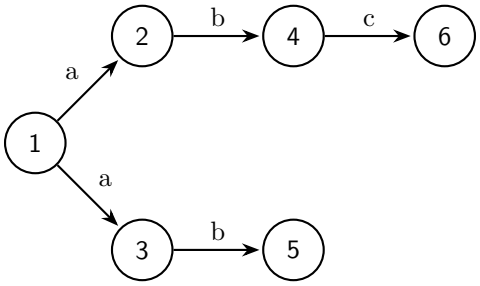


Figure 2.6: Example graph $\mathcal{G}_{ex}$.

2.2.5 k-path-bisimulation

As mentioned in Section 2.1.1, the notion of k-path-bisimulation is used for partitioning in the path based language-aware index [65]. Using this notion and the general idea behind a language-aware index introduced in Section 2.2.4, we can partition the paths in a graph into blocks such that all queries in CPQ_k either match all paths in a block or none at all. Next we will formally present a recursive definition for k-path-bisimulation on a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ with label set $\mathcal{L}$, edge set $\mathcal{E}$ and vertex set $\mathcal{V}$. This definition can also be found in the language-aware indexing paper and more information on the concept of bisimulation can be found in the following papers by Fletcher et al. [24, 25]. For two source target pairs (v, u) and (x, y), the pairs are k-path-bisimilar $(v, u) \approx_k (x, y)$ if and only if:

1) $v = u$ if and only if $x = y$. Two paths can only be indistinguishable with respect to *CPQ* if they are either both cycles or both not cycles. Otherwise the identity operation can be used to distinguish these paths.

2) If $k > 0$ then for each label $l \in \mathcal{L}$ two conditions have to hold:

  1) If $(v, u, l) \in \mathcal{E}$ then also $(x, y, l) \in \mathcal{E}$ and similar for reverse edges, if $(u, v, l) \in \mathcal{E}$ then also $(y, x, l) \in \mathcal{E}$.

  2) If $(x, y, l) \in \mathcal{E}$ then also $(v, u, l) \in \mathcal{E}$ and similar for reverse edges, if $(y, x, l) \in \mathcal{E}$ then also $(u, v, l) \in \mathcal{E}$.

  Essentially, if one of the pairs is connected with a path of length 1, then the other pair also needs to be connected with a path of length 1. Furthermore, these paths need to share the same label.

3) If $k > 1$ then we again have two conditions:

  1) For all $m \in \mathcal{V}$, if $(v, m) \in \mathcal{P}^{\leq k-1}$ and $(m, u) \in \mathcal{P}^{\leq k-1}$, then there exists some $m' \in \mathcal{V}$ such that $(x, m') \in \mathcal{P}^{\leq k-1}$ and $(m', y) \in \mathcal{P}^{\leq k-1}$. Moreover, $(v, m) \approx_{k-1} (x, m')$ and $(m, u) \approx_{k-1} (m', y)$.

  2) For all $m \in \mathcal{V}$, if $(x, m) \in \mathcal{P}^{\leq k-1}$ and $(m, y) \in \mathcal{P}^{\leq k-1}$, then there exists some $m' \in \mathcal{V}$ such that $(v, m') \in \mathcal{P}^{\leq k-1}$ and $(m', u) \in \mathcal{P}^{\leq k-1}$. Moreover, $(x, m) \approx_{k-1} (v, m')$ and $(m, y) \approx_{k-1} (m', u)$.

  With these conditions we are essentially saying the following, if one of the pairs can be constructed from two paths of at most length $k - 1$ then this must also hold for the other pair. Furthermore, the paths used to construct each pair are $(k - 1)$-path-bisimilar.

Intuitively two paths are k-path-bisimilar when any possible sequence of label traversals of at most length k can be performed from both source vertices v and x to end up at target vertices u and y respectively. Essentially this means that the only difference between these paths is the identity of the traversed vertices, while all traversed edges have the same labels.

As we have reiterated a number of times, there is a strong connection between the notion of k-path-bisimulation and the expressive power of CPQ_k. In fact, using k-path-bisimulation as a structural characterisation of the expressive power of CPQ_k is what allows us to use it to construct the equivalence class blocks in Section 4.1. Thus it is relevant to restate Theorem 4.1 from the language-aware indexing paper which originally formally established this relation [65, 24].

Theorem 1. *Let $\mathcal{G}$ be a graph, k be a non-negative integer, and $v, u, x, y \in \mathcal{V}$. If $(v, u) \approx_k (x, y)$, then for every $q \in CPQ_k$ it holds that $(v, u) \in [\![q]\!]_{\mathcal{G}}$ if and only if $(x, y) \in [\![q]\!]_{\mathcal{G}}$.*

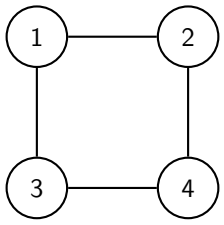
Informally this theorem states that if two paths are k-path-bisimilar, then if one of them appears in the evaluation result of a *CPQ* the other one also has to appear in the result. Note that due to the bi-implication this also means that if one of the paths does not appear in the result, the other path will not appear in the result either.

2.2.6 Homomorphism and Isomorphism

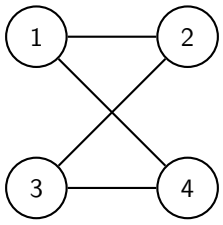
A graph homomorphism is a structure preserving mapping between two graphs. Essentially this means that a graph homomorphism can be used to check if a graph captures all the important structure of a different graph. Formally, given two graphs, $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1, \mathcal{L})$ and $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2, \mathcal{L})$ and assuming that there is a homomorphism f from $\mathcal{G}_1$ to $\mathcal{G}_2$ denoted as $f : \mathcal{G}_1 \to \mathcal{G}_2$, then this implies that $\forall_{u,v \in \mathcal{V}_1 \wedge l \in \mathcal{L}} [(u, v, l) \in \mathcal{E}_1 \implies (f(u), f(v), l) \in \mathcal{E}_2]$. In other words, a homomorphism is a mapping from the vertices of a graph to the vertices of another graph. Moreover, this mapping is constructed in such a way that two adjacent vertices in the graph are still adjacent in the other graph after being mapped, meaning the structure of the graph is preserved. Also note that any edge labels are required to match for the edges as they are implicitly being mapped when their end points are mapped. If this relation between graphs holds in both directions, that is both $\mathcal{G}_1 \to \mathcal{G}_2$ and $\mathcal{G}_2 \to \mathcal{G}_1$ hold, then $\mathcal{G}_1$ and $\mathcal{G}_2$ are called homomorphically equivalent.

Note that this definition allows the same vertex to be mapped onto by multiple vertices. This is the key difference between a homomorphism and an isomorphism, an isomorphism is a homomorphism that is also bijective. Thus, in an isomorphism there is a one to one mapping between the vertices in the two graphs and all the vertices from both graphs participate in this mapping. Consequently, this also implies that the number of vertices as well as the number of edges is the same in both graphs. In fact, two isomorphic graphs cannot structurally be distinguished at all unless there is additional information available, such as vertex labels.

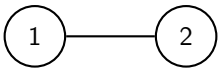
To give some examples, consider the undirected and unlabelled graphs in Figure 2.7, the vertex labels are purely for illustrative purposes so we can refer to vertices and should not be considered as part of the graph structure. Starting with graph $\mathcal{G}_1$ in Figure 2.7a, note that this graph is isomorphic to the graph $\mathcal{G}_2$ in Figure 2.7b since we can map the vertices as follows $1 \mapsto 1$, $2 \mapsto 2$, $3 \mapsto 4$, and $4 \mapsto 3$. However, also note that both $\mathcal{G}_1$ and $\mathcal{G}_2$ are homomorphic to $\mathcal{G}_3$ in Figure 2.7c. For example a possible homomorphism mapping from $\mathcal{G}_1$ is $1 \mapsto 1$, $2 \mapsto 2$, $3 \mapsto 2$, and $4 \mapsto 1$.



(a) Example graph $\mathcal{G}_1$.



(b) Example graph $\mathcal{G}_2$.



(c) Example graph $\mathcal{G}_3$.

Figure 2.7: Example graphs for showing homomorphism and isomorphism.

2.2.7 Graph Cores

Building on the definitions from Section 2.2.6, the core of a graph is the smallest homomorphically equivalent subgraph and is also unique up to isomorphism [32]. Essentially, a graph core captures all of the important information from a larger graph. Formally, if $\mathcal{G}_2 \subseteq \mathcal{G}_1$ and $\neg\exists_{(\mathcal{G}_i=(\mathcal{V}_i,\mathcal{E}_i,\mathcal{L}))\subseteq\mathcal{G}_1} [|\mathcal{V}_i| < |\mathcal{V}_2| \wedge \mathcal{G}_1 \to \mathcal{G}_i \wedge \mathcal{G}_i \to \mathcal{G}_1]$, then $\mathcal{G}_2$ is the core of $\mathcal{G}_1$. In other words, if $\mathcal{G}_2$ is a smallest homomorphically equivalent subgraph of $\mathcal{G}_1$, then

$\mathcal{G}_2$ is the core of $\mathcal{G}_1$. Furthermore, it is worth noting that two graphs are homomorphically equivalent if and only if their cores are isomorphic. Combined with the property that graph cores are unique up to isomorphism, this makes them ideal candidates as representatives for an equivalence class. Note that Figure 2.7c is the core of both Figure 2.7a and Figure 2.7b as no smaller homomorphically equivalent graphs exist.

2.2.8 Tree Decompositions and Treewidth

A tree decomposition of a graph is a tree where each tree node contains a number of vertices from the original graph that together represent a connected subgraph. The reason we are interested in tree decompositions is that they can often be used to solve problems more efficiently, and we will use them extensively in Chapter 3. By solving problems on the tree decomposition of a graph instead of the graph itself, we can take advantage of the fact that many problems are trivial or easy to solve on a tree. Tree decompositions are often accompanied by a metric called treewidth, which essentially indicates how far removed from a tree the original graph was, with trees having a treewidth of 1. Graphs with a low treewidth often admit very efficient solutions to problems that are hard or intractable in the general case. Conveniently, based on previous research [26, 20] we can state the following about the treewidth of *CPQ*s.

Theorem 2. CPQ*s have a treewidth of 2.*

Interestingly, despite their current popularity, treewidth and tree decompositions have a long history of constantly being rediscovered. A metric equivalent to treewidth called the dimension of a graph was originally proposed in 1972 [7], then later reintroduced in 1976 together with the concept of a tree decomposition [31], only for both to be rediscovered again in 1984 under their current names [61]. While originally treewidth was only defined for undirected graphs, it was later generalised to directed graphs [41]. It is also worth noting that many graph related metrics were later discovered to be equivalent to treewidth [10, 40]. Notable for *CPQ*s is that graphs with treewidth at most k are k-degenerate [30], implying $|\mathcal{E}| \leq k \cdot |\mathcal{V}|$, which for *CPQ*s becomes $|\mathcal{E}| \leq 2 \cdot |\mathcal{V}|$, giving us an upper bound on the number of edges in a *CPQ*.

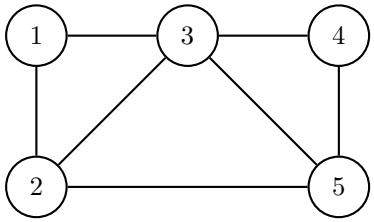
Formally, a tree decomposition has to adhere to three rules, in these rules the nodes or bags of the tree decomposition are given by $\mathcal{X} = \{X_1, \ldots, X_n\}$. Note that each of these bags or subsets is a subset of the vertex set $\mathcal{V}$ of the original graph.

1) All vertices from the vertex set $\mathcal{V}$ appear in at least one set in $X \in \mathcal{X}$, implying that the union of all subsets equals the vertex set $X_1 \cup \cdots \cup X_n = \mathcal{V}$.

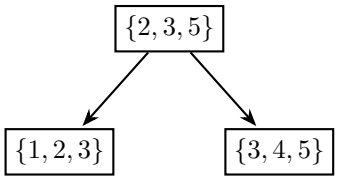
2) Both end point vertices of every edge $(u, v) \in \mathcal{E}$ appear together in at least one subset $X \in \mathcal{X}$, formally this means $\exists_{X \in \mathcal{X}}[(u, v) \in \mathcal{E} \implies u \in X \wedge v \in X]$.

3) Each vertex $v \in \mathcal{V}$ induces a connected subtree, essentially this means that all the subsets $X \in \mathcal{X}$ that contain v are connected such that they can be reached from each other by only visiting other bags that also contain v.

Note that this means that the tree decomposition of a graph is not necessarily unique. In fact, putting all the nodes of a graph in the same node in the tree decomposition is a valid tree decomposition. However, such a tree decomposition is of little use, so instead we often want to find the tree decomposition with the smallest bag sizes. The size of the largest bag minus one is then the treewidth of the tree decomposition. An example of a graph and a possible tree decomposition for it is given in Figure 2.8. Note that this tree decomposition has a treewidth of 2.



(a) Example graph.



(b) A tree decomposition of the example graph.

Figure 2.8: Example showing a graph and a possible tree decomposition for it of treewidth 2.

2.3 Reference Implementation

This thesis comes with two completely documented reference implementations written primarily in Java. As noted in Section 2.1.4, more universally applicable features were added to gMark and the implementation of the index itself was released separately. Next we will briefly cover each codebase in more detail and state where each major algorithm is implemented. Note that the following sections will reference algorithms that will only be discussed in Chapter 3 and 4 of this thesis.

2.3.1 gMark Extensions

Most more broadly applicable *CPQ* related utilities have been implemented in gMark, and are available starting from release 1.2 [37]. The gMark repository[1] is released publicly on GitHub under the GPL v3.0 license [28]. Instructions for getting started with gMark are available in Appendix B. The key features implemented during the thesis project are:

1) ***CPQ* Cores**: *CPQ* core computation based on the algorithm presented in Section 3.3.
2) **Query Homomorphism**: The *CPQ* query homomorphism testing method presented in Section 3.1.3.
3) **Tree Decomposition**: The tree decomposition algorithm from Section 3.1.2.
4) **CPQ API Extensions**: Most notably the ability to parse text form *CPQ*s, more functions for easy *CPQ* construction, and exposing more information about constructed *CPQ*s and *CPQ* query graphs. Visualising constructed *CPQ*s was also made easier.
5) **New Data Structures**: Implementation of data structures for simple high performance graphs and trees.
6) **Various Utilities**: A number of miscellaneous utility algorithms have been introduced for working with graphs, computing maximal graph matchings, computing the Cartesian product of sets, and computing all subsets of a set.

Less notable smaller features were also introduced, often to support some of the more extensive algorithms.

2.3.2 *CPQ*-native Index

The implementation of the *CPQ*-native Index that will be discussed in Chapter 4 and the canonisation logic from Section 3.4 are released as a separate repository that has the gMark repository as a dependency. This repository[2] is available as open-source on GitHub [36] under the GPL v3.0 license [28]. Instructions for using the index software can be found in Appendix C.

[1] `https://github.com/RoanH/gMark`
[2] `https://github.com/RoanH/CPQ-native-index`

The Core of a *CPQ*

3 第三章

As mentioned in Section 1.1, the simplest way to think of a database index is that of a data structure where information relevant to query evaluation can be retrieved using some key. Naturally, this means that the exact definition of this key is extremely important to the design of an index. For the *CPQ*-native Index that will be described in Chapter 4 of this thesis, the key will be the canonical form of a *CPQ* core. The goal of this chapter is to formalise the exact definition of a *CPQ* core and to show how the core of a *CPQ* and its canonical form can be computed.

Within this chapter we will first introduce an important query equivalence test in Section 3.1 that is required by the algorithm for core computation . This test will make use of an augmented definition for homomorphism and a new algorithm for computing tree decompositions. Using this test we will then formalise the definition of a *CPQ* core in Section 3.2 and show an intuitive but naive algorithm for computing cores. In Section 3.3 we will then present a more complex but also significantly more efficient algorithm for computing cores. Next in Section 3.4 we will present an approach to computing a canonical representation of a *CPQ* core. Finally, in Section 3.5 we will briefly summarise the content of this chapter.

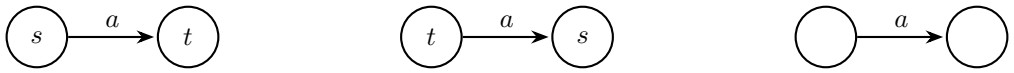
3.1 Testing Query Equivalence

Although the exact definition of a *CPQ* core will be formalised in Section 3.2, it is worthwhile to already mention some key properties. Similar to the concept of a graph core explained in Section 2.2.7, the core of a *CPQ* is a smaller subgraph of the query graph of the *CPQ*. However, what makes a *CPQ* core interesting is the fact that its evaluation result on any graph is equivalent to the evaluation result of the original query. In other words, the output for the original *CPQ* and its core are exactly the same. Note that this is not necessarily true for the regular graph core of a *CPQ* as we will demonstrate in Section 3.1.1.

The focus of this section will be on testing if the evaluation result of a *CPQ* is contained in the evaluation result of another *CPQ*, meaning the results returned by the query are a subset of the results returned by the other query. This test will form the basis for the method to compute *CPQ* cores we will introduce in Section 3.2 and also serve as the core inspiration behind the more efficient algorithm for core computation we will discuss in Section 3.3. Finally, note that if this containment relation holds in both directions, then the queries are equivalent.

3.1.1 Query Homomorphism

In Section 2.2.6 we introduced the notion of a homomorphism as a structure preserving mapping between the vertices of two graphs. The goal of this section is to extend the notion of a homomorphism to that of a query homomorphism, which we can use in Section 3.2 to formalise the definition for the core of a *CPQ*. However, first it is worth demonstrating why the definition of a regular graph core based on a regular homomorphism as introduced in Section 2.2.7 is not a suitable definition of a *CPQ* core using a simple example. In Figure 3.1 the query graphs for the *CPQ*s $q_1 = a$ and $q_2 = a^-$ are shown in Figure 3.1a and Figure 3.1b respectively.



(a) Query graph $\mathcal{G}_{q_1}$ of $q_1 = a$. (b) Query graph $\mathcal{G}_{q_2}$ of $q_2 = a^-$. (c) Graph core of $\mathcal{G}_{q_1}$ and $\mathcal{G}_{q_2}$.

Figure 3.1: Example showing that a graph core is not a *CPQ* core.

Observe that the core presented in Figure 3.1c is in fact a graph core of both query graphs. However, the original *CPQ*s are not equivalent. The main issue is that a regular graph core does not consider the special nature of the source and target vertices in a *CPQ* query graph. Essentially, these vertices are labelled and thus should never be allowed to be mapped to each other, unless they are actually the same vertex.

Fortunately, this is relatively easy to resolve as we just need to extend the definition of a regular homomorphism to respect the special nature of the source and target vertex. We first start with the original definition of a homomorphism, recall that for two graphs $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1, \mathcal{L})$ and $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2, \mathcal{L})$ we have that $\forall_{u,v \in \mathcal{V}_1 \wedge l \in \mathcal{L}} [(u, v, l) \in \mathcal{E}_1 \implies (f(u), f(v), l) \in \mathcal{E}_2]$, meaning edges adjacent in $\mathcal{G}_1$ also have to be adjacent in $\mathcal{G}_2$ via edges with identical edge labels. Assuming both graphs are query graphs and that $s_1, t_1 \in \mathcal{V}_1$ and $s_2, t_2 \in \mathcal{V}_2$ are the source and target vertices of these query graphs, we extend the definition of a homomorphism with the requirements that $f(s_1) = s_2$ and $f(t_1) = t_2$, meaning the source vertex and target vertex have to be mapped to the source vertex and target vertex respectively. This resolves our issue by essentially fixing part of the homomorphism mapping and we will call this augmented definition a query homomorphism.

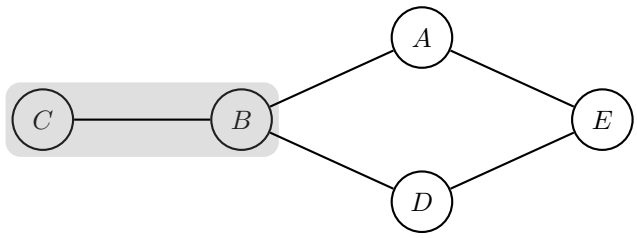
3.1.2 Computing Tree Decompositions of Treewidth 2

For part of the algorithm that will be described in Section 3.1.3 we need to be able to compute a tree decomposition of an input graph that has a treewidth of at most 2. A prime candidate for this task is an algorithm from a well known paper by Hans Bodlaender [9, 11]. This paper shows a linear time algorithm that for a fixed value of k can compute a tree decomposition of width at most k given an input graph with treewidth at most k. However, this algorithm is complex and has a relatively large hidden constant in its runtime. Furthermore, it handles any fixed treewidth, while we only need an algorithm for a treewidth of 2. Hence, we will describe a newly developed linear time algorithm in this section for finding tree decompositions of graphs with a treewidth that is at most 2. However, note that while the algorithm is newly developed, it is not really novel. The algorithm that will be discussed in this section is based on an algorithm featured in one of the practise problems by the International Collegiate Programming Contest Japanese Alumni Group (ICPC OB/OG) in 2012 [2, 22]. One key difference is that the algorithm featured here never has to deal with vertices of degree 1. It is also worth noting that this algorithm is likely inspired by a well known paper on the recognition of series-parallel (*SP*) digraphs, which features the same reduction rule [68]. Recall from Section 2.1 that *CPQ* is an extension of *SP*. Unfortunately, this extension causes the reduction rules presented in this paper to be insufficient for *CPQ*s. However, the algorithm presented in this section does borrow the same reduction rule as the ICPC competition.

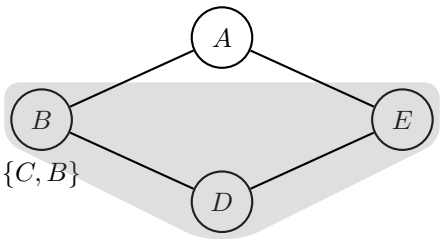
Next follows a complete description of the algorithm to compute a width 2 tree decomposition from an input graph with treewidth at most 2. Similar to its predecessors, this algorithm uses reduction rules and repeatedly reduces vertices of degree at most 2. Each reduction rule used in the algorithm reduces the total number of vertices in the graph by one and generates a tree decomposition node with either two or three vertices. These intermediary tree decomposition nodes are attached as metadata to edges and vertices that remain in the graph. When the graph has at most 3 vertices remaining the algorithm stops and computes the final tree decomposition. The following two reduction rules are used depending on the degree of the vertex being reduced:

- **Degree 1**: Degree 1 vertices are reduced by first creating a tree decomposition node with the vertex and the one vertex it is adjacent to. The degree 1 vertex and its edge are then removed from the graph and the information for the decomposition node is attached as metadata to the vertex the removed vertex was adjacent to. If the removed vertex or the edge being removed had attached intermediary tree decomposition nodes these nodes are attached as child nodes of the node that was newly created. An example of this reduction rule being applied can be seen in Figure 3.2, where Figure 3.2a shows the original graph and highlights the edge and vertex involved in the reduction and Figure 3.2b shows the graph after vertex C was removed and the tree decomposition node $\{C, B\}$ attached to vertex B.
- **Degree 2**: Degree 2 vertices are reduced following the borrowed reduction rule from the ICPC competition. First a tree decomposition node is created using the vertex and the two vertices adjacent to it. The degree 2 vertex and its edges are then removed from the graph and the decomposition node is attached as metadata to the edge between the two vertices the removed vertex was adjacent to, if this edge does not exist it is added to the graph. If the removed vertex or edges had any attached intermediary tree decomposition nodes these are attached as child nodes of the newly created node. An example of this reduction rule being applied can be seen in Figure 3.2, where Figure 3.2b shows the original graph and highlights the edges and vertices involved in the reduction and Figure 3.2c shows the graph after vertex D was removed and the tree decomposition node $\{B, D, E\}$ added to the new edge between B and E.

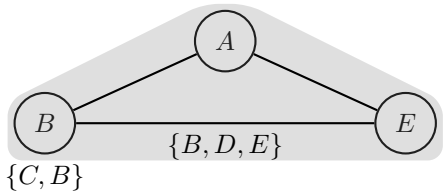
Finally, the remaining vertices form the root node of the tree decomposition. Any intermediary tree decomposition nodes on the remaining vertices and edges become children of this root node. An example of this step is shown in Figure 3.2, where Figure 3.2c shows the final graph and 3.2d the constructed tree decomposition.



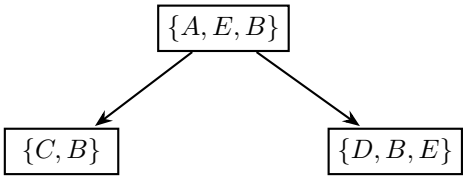
(a) Reduction of the degree 1 vertex C.



(b) Reduction of the degree 2 vertex D.



(c) Final reduction of the graph.



(d) The constructed tree decomposition.

Figure 3.2: Example showing the construction of a tree decomposition using reduction rules.

Note that this construction does not guarantee much about the so called quality of the created tree decomposition. Generally we call a tree decomposition 'nice' when the following conditions apply to the nodes in the tree decomposition [44]:

1) All tree decomposition nodes have at most 2 children.

2) If a node has exactly one child node, then it differs from the child node by exactly 1 vertex.

3) If a node has two child nodes, then the bags for these child nodes and the parent node contain exactly the same vertices.

When a tree decomposition has these properties it becomes possible to classify all the tree decomposition nodes as either start, join, forget, or introduce nodes. These special node designations make it easier to design dynamic programming algorithms. Note that our algorithm unfortunately does not offer any guarantees like this. While each tree decomposition node has at most 3 vertices in it, the created tree may not be balanced and a single node can have arbitrarily many direct child nodes. We also do not provide the listed guarantees about the number of differing vertices between two adjacent nodes. However, do note that we have a bound on the difference. Bags created by a degree 1 reduction share exactly one vertex with their parent node, and bags created by a degree 2 reduction share exactly two vertices with their parent node. Fortunately, it is worth noting that there are existing algorithms for turning an existing width k tree decomposition into a nice tree decomposition of width k in linear time [44].

Next it is worth noting that the created tree decomposition is likely far from minimal as for a graph with $|\mathcal{V}|$ vertices the tree decomposition has $1 + \max(0, |\mathcal{V}| - 3)$ nodes. It is also worth noting that there are restrictions on exactly what input graphs the algorithm can handle, though some of the restrictions we will list could be supported relatively easily. The following is a list of assumptions that are made by the algorithm:

1) The input graph has a treewidth of at most 2.

2) The input graph is connected.

3) The input graph contains no self loops.

4) The input graph contains no parallel edges.

Observe that these restrictions mean that this version of the algorithm does not apply to *CPQ*s directly. Nevertheless, for our intended use case this is not an issue as we do not need to directly compute the tree decomposition of a *CPQ* query graph. However, it is worth noting that the problematic 3rd and 4th item mainly present a counting problem where the degree of a vertex is inflated, modifying the algorithm to accommodate these two graph patterns is relatively easy. Self loops can simply be removed and parallel edges can be collapsed into a single edge. The full pseudocode for the described algorithm is given in Algorithm 1.

Algorithm 1 Computing a tree decomposition of a graph with treewidth at most 2.

procedure ComputeTreeDecomp($\mathcal{G} = (\mathcal{V}, \mathcal{E})$) ▷ Edges and vertices have an initially empty *trees* set.
  $\mathcal{Q} \leftarrow \{v \mid v \in \mathcal{V} \wedge \deg(v) \leq 2\}$ ▷ Collect vertices of degree at most 2.
  while $|\mathcal{V}| > 3$ **do** ▷ The root is a bag with at most 3 vertices.
    $v \in \mathcal{Q}$
    if $\deg(v) = 1$ **then**
      $e \in v.edges$ ▷ The only edge v is an end point of.
      $t \in e \wedge t \neq v$ ▷ The other end point of the edge.
      $t.trees \leftarrow t.trees \cup \{\{v, t\}, v.trees \cup e.trees\}$ ▷ Add a tree node with the given *bag* and *children*.
      $\mathcal{G} \leftarrow \mathcal{G} \setminus \{v\}$ ▷ Delete v from the graph together with all its edges.
      if $\deg(t) = 2$ **then**
        $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{t\}$
      end if
    else ▷ Degree is 2.
      $e_1, e_2 \in v.edges \wedge e_1 \neq e_2$
      $v_1 \in e_1 \wedge v_1 \neq v$
      $v_2 \in e_2 \wedge v_2 \neq v$
      $n \leftarrow$ **false**
      $e \in v_1.edges \wedge v_2 \in e$
      if $e =$ **nil then** ▷ Reuse existing edges if present.
        $n \leftarrow$ **true**
        $e \leftarrow \{v_1, v_2\}$
        $\mathcal{G} \leftarrow \mathcal{G} \cup \{e\}$ ▷ Add a new edge to the graph.
      end if
      $e.trees \leftarrow e.trees \cup \{\{v, v_1, v_2\}, v.trees \cup e_1.trees \cup e_2.trees\}$
      $\mathcal{G} \leftarrow \mathcal{G} \setminus \{v\}$
      if $\neg n$ **then** ▷ Only check degrees if the edge was not newly added.
        if $\deg(v_1) = 2$ **then**
          $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{v_1\}$
        end if
        if $\deg(v_2) = 2$ **then**
          $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{v_2\}$
        end if
      end if
    end if
    $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{v\}$
  end while
  return $\mathcal{T} \leftarrow \{\mathcal{V}, (\bigcup_{e \in \mathcal{E}} e.trees) \cup (\bigcup_{v \in \mathcal{V}} v.trees)\}$ ▷ All remaining vertices form the root node.
end procedure

To conclude, we will discuss the correctness of the algorithm and its runtime.

Theorem 3. *Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a tree decomposition $\mathcal{T}$ with bags $\mathcal{X} = \{X_1, \ldots, X_n\}$ created by Algorithm 1, $\mathcal{T}$ is a valid tree decomposition of $\mathcal{G}$ if $\mathcal{G}$ was a valid input graph for the algorithm.*

Proof. Recall from Section 2.2.8 that any valid tree decomposition has to satisfy three properties. In order to prove correctness we will separately prove each of these properties:

1) The first property states that each vertex $v \in \mathcal{V}$ has to be contained in at least one bag $X_i \in \mathcal{X}$. Note that this is trivially the case for vertices that were reduced, as a new bag with these vertices is created when they are deleted. Furthermore, the tree decomposition nodes stored at any edge and node are inherited when they are deleted, so all bags end up in the final tree decomposition. Finally, note that any remaining nodes are assigned to the root bag of the tree decomposition. Therefore, all vertices are in a bag after the algorithm finishes.

2) The second property states that for each edge $(u, v) \in \mathcal{E}$ there has to be a bag $X_i \in \mathcal{X}$ containing both u and v. Note that this proof will be similar to the first property. When a vertex is reduced all the vertices adjacent to it are placed in the bag that was newly created, thus the edges that were connected to the vertex are covered by this bag. Following the same logic as for the proof of the first property, these bags are inherited when their carrying edge or vertex is deleted. Finally, note that any edges that still remain at the end must be between the final at most 3 vertices of the graph. Since these vertices are placed in the root together these edges are covered as well.

3) The last property states that if $v \in X_i$ and $v \in X_j$ with $X_i, X_j \in \mathcal{X}$, then any tree node on the path from X_i to X_j has to contain v. Note that when v is contracted, its bag is attached to either the only vertex it was adjacent to or the edge between the two vertices it was adjacent to. In both cases the presence of v in the graph ends, meaning it will not be part of any bags created later on. For the proof this means we are primarily interested in the part of the tree decomposition rooted at the bag v was contracted to. For the contraction to a vertex case it is easy to see that any bags that were directly attached to v had to contain v, since the other vertex in these bags had to be adjacent to v. For the case where the contraction is to an edge, note that the only bags that could contain v would need to be attached to the edges directly adjacent to v. Since these bags become children of the same node when v is contracted, the v induced subgraph remains connected. Finally, we consider the case where an edge adjacent to v is involved in an contraction. Note that edges like this may have an attached bag that contains v. However, note that for a contraction like this v itself has to also be a part of the constructed bag as otherwise the edge cannot be involved. Thus also in this case the v induced subgraph remains connected. Therefore, the final property also holds.

Since each of the three required properties holds, the algorithm generates a valid tree decomposition. □

Theorem 4. *Algorithm 1 can be implemented to run in linear time* $\mathcal{O}(|\mathcal{V}|)$.

Proof. Note that the algorithm makes use of a list to track vertices of degree at most 2 that can be processed. Furthermore, this list is kept updated in constant time, this design element is key to an implementation of the algorithm that runs in linear time. For the analysis of the runtime we will focus on the parts with a runtime that is not obviously bounded in the number of vertices. This means that we will skip lines like the first line of the algorithm that simply scan the list of vertices.

The most interesting part of the algorithm is the main while loop on line 3. Note that this while loop runs at most $\mathcal{V} - 3$ times as both of the two main branches inside it reduce the number of vertices by 1 on line 9 and 25 respectively. Thus to prove that the loop as a whole runs in at most linear time it remains to prove that all the statements in the body of the while loop run in constant time. Fortunately, most operations are trivially constant time as they either select random elements from a set, extend a set by a single element, or perform a simple check. Note that the deletion of a vertex is constant time since the vertices that are being removed are of degree at most 2, meaning the number of adjacent edges that need to be removed is bounded. Similarly, note that the sets being taken from on lines 6, 7, 14, 15 and 16 are similarly all of size at most 2, making these statements constant time.

This means that the main problematic statements are the set unions on line 8 and 24. Note that we already claimed before that a single element could have an arbitrary number of metadata nodes attached to it, trivially this means that these lines could never take constant time. However, we can bound the runtime when considering the algorithm as a whole. Note that the purpose of these set unions is to attach child nodes of the tree decomposition to their parent. Naturally each tree decomposition node can only be attached to a single parent and they are only attached once as their metadata carrying element is deleted from the graph afterwards. This means that we want to find a bound on the total number of times a child node is attached to its parent node. Note that this is equivalent to finding a bound on the total number of tree decomposition nodes created as each node will only ever be attached to a single parent. Recall that we already presented a bound for this when stating that the created tree decomposition would not necessarily be a nice one. Since at most $|\mathcal{V}| - 3$ reductions are carried out, and thus at most $\max(0, |\mathcal{V}| - 3)$ nodes are created this way, this also means that the number of tree decomposition bags we created is linear in the number of vertices. Thus, across the entire execution of the algorithm these set unions take linear time in the number of vertices.

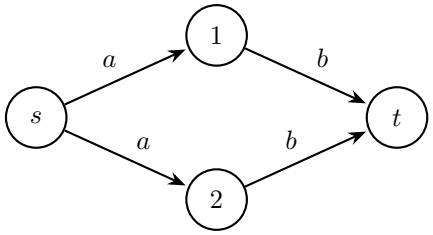
For the final construction of the root tree decomposition node on line 37, note that iterating all edges is bounded by the number of remaining vertices squared. Since at most 3 vertices remain, this line is bound by a constant. For the union of all the sets, note that the same logic from the previous paragraph applies. However, as an alternative bound we could use the fact that the number of tree decomposition nodes is already bounded by the number of vertices. To conclude, since all parts of the algorithm run in $\mathcal{O}(|\mathcal{V}|)$, the claimed runtime holds. □

To conclude, the algorithm we have described is both correct and has a linear runtime. Finally, it is worth reiterating that there are minor modifications that can be made to increase the classes of graphs the algorithm is applicable to.

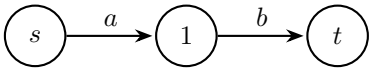
3.1.3 Conjunctive Query Containment

The main idea for the approach that will be discussed in this section comes from a paper on conjunctive query containment by Chandra Chekuri and Anand Rajaraman [17]. The approach we will present is effectively a concrete instantiation that mostly follows this paper with a few important modifications to suit our use case.

In the following sections we will make use of a running example using the *CPQ* $q = (a \circ b) \cap (a \circ b)$, also shown in Figure 3.3a. Note that this *CPQ* contains the conjunction of two identical paths $a \circ b$. As such, it is easy to see that the query is query homomorphic to the *CPQ* $q' = (a \circ b)$, also shown in Figure 3.3b. For both query graphs, nodes that are not the source or target vertices have been given arbitrary labels to tell them apart in the examples that will follow later. Note that these labels are purely for illustrative purposes as *CPQ*s do not have vertex labels.



(a) Running example query graph for $q = (a \circ b) \cap (a \circ b)$.



(b) Query graph for $q' = a \circ b$.

Figure 3.3: Running example *CPQ*s for conjunctive query containment.

Before describing the algorithm, we will first introduce the concept of a conjunctive query (*CQ*). As mentioned in Section 2.1, *CPQ* is a subset of *CQ*, thus any *CPQ* is also a valid *CQ*. Formally, a *CQ* is a query using only the logical conjunction operator ($\wedge$) and atomic formulae. However, in the context of a graph database it is easier to think of a *CQ* as a set of edge predicates [12]. Each of these edge predicates matches a single edge in the graph and all predicates have to match at the same time for it to be matched by the entire *CQ*. In the context of a relational database we would effectively have a table named after each edge label with a source and target vertex column in each table. The general form of a *CQ* is as shown in Equation 3.1.

$$(z_1, \ldots, z_m) \leftarrow a_1(x_1, y_1), \ldots, a_n(x_n, y_n) \tag{3.1}$$

For a graph with vertex set $\mathcal{V}$, label set $\mathcal{L}$ and $1 \leq i \leq n$, we have $x_i \in \mathcal{V}$, $y_i \in \mathcal{V}$ and $a_i \in \mathcal{L}$. Furthermore, for $1 \leq j \leq m$, we have $z_j \in (\{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_n\})$, meaning that all head variables need to appear in the body of the query. However, it is allowed for the arity of the query m to be 0, this makes it a boolean query. Since the arity of a *CPQ* is 2, this will be the main arity we deal with in this thesis. Knowing how a *CQ* works we can now write the *CQ* form of the *CPQ* in Figure 3.3a. Note that the query graph for this *CPQ* has 4 edges, thus our *CQ* will have 4 predicates. The *CQ* form of the query is shown in Equation 3.2.

$$(s, t) \leftarrow a(s, 1), a(s, 2), b(1, t), b(2, t) \tag{3.2}$$

Similarly, the *CQ* form of the query graph in Figure 3.3b is given in Equation 3.3.

$$(s, t) \leftarrow a(s, 1), b(1, t) \tag{3.3}$$

3.1.3.1 The Incidence Graph of a Query Graph

The first step towards implementing the algorithm from the paper is to compute the incidence graph of the input query [17]. The incidence graph of a query is a bipartite undirected graph with a node for each vertex and edge in the original graph. One partition of the bipartite graph consists of all former vertices and the other partition consists of all former edges. Two nodes are linked by an edge if the vertex node is one of the end points of the edge in the edge node. Since this requires the *CQ* form of our input query, we will be using the expression shown in Equation 3.2. The incidence graph for the running example *CPQ* is then as shown in Figure 3.4. Note that incidentally all nodes have degree 2, while this will always be the case for edge nodes, there is no bound on the number of edges that can be attached to a vertex node.

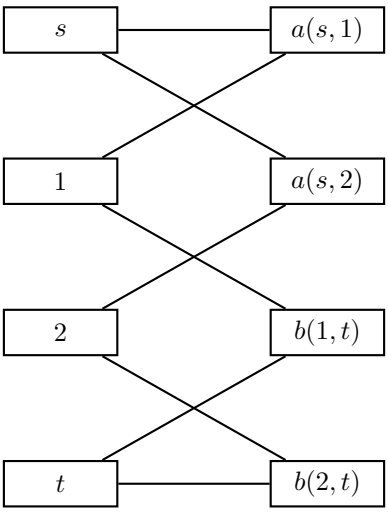


Figure 3.4: The incidence graph for the running example.

One final important remark is that the treewidth of the created incidence graph is at most the treewidth of the original graph [17], this fact will be important for the tree decomposition we will compute in Section 3.1.3.2.

3.1.3.2 An Incidence Graph Tree Decomposition

After computing the incidence graph of the input graph the next step is computing a tree decomposition of the incidence graph. For this purpose we will use the tree decomposition algorithm developed in Section 3.1.2. Note that the incidence graph satisfies all the assumptions made by this algorithm and that we established in Section 3.1.3.1 that the treewidth of the incidence graph is at most 2. A tree decomposition of the incidence graph for the running example shown in Figure 3.4 is given in Figure 3.5.

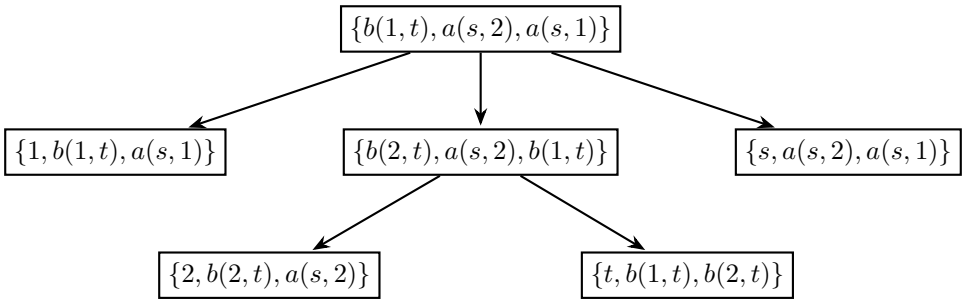


Figure 3.5: The tree decomposition of the incidence graph for the running example.

It is worth noting that in this particular case the incidence graph is actually a simple cycle. Thus using 6 bags for the tree decomposition is actually minimal.

3.1.3.3 Partial Mappings between Graphs

The next step of the procedure is the most important part and involves constructing partial mappings between the two graphs being tested for query homomorphism. The goal of a partial mapping is to identify smaller parts of the input graphs that are query homomorphic to each other. For the approach we will cover, these smaller parts are either vertices or edges. It is also worth noting that this section features our main deviation from the conjunctive query containment paper [17]. The reason for this is that we need to ensure that our partial mappings satisfy the definition of a query homomorphism instead of the definition of a regular homomorphism. In order to do this, we need to add extra rules to properly handle source and target vertices in our mappings.

We will start by introducing the simplest way of constructing partial mappings between our input graphs. Note that to do this we only need to ensure that the definition of a query homomorphism is met for each partial mapping. Since a query homomorphism only states something about edges, it would be sufficient to just map two edges exactly following the constraints in the definition of a query homomorphism as given in Section 3.1.1. Based on this idea, when given two edges $e = (u, v, l)$ and $e' = (u', v', l')$, a partial mapping $e \mapsto e'$ exists if the following rules hold:

1) If u is a source vertex then u' has to be a source vertex.

2) If v is a target vertex then v' has to be a target vertex.

3) Edge labels match: $l = l'$.

Using these rules we can determine the partial maps for our running example *CPQ*s q and q' from Figure 3.3. Note that the left hand side has elements from q while the right hand side has elements from q'. In addition, we use a set for the right hand side of our maps to indicate all the possible valid mapping targets.

- $s \mapsto \{s, 1, t\}$
- $1 \mapsto \{s, 1, t\}$
- $2 \mapsto \{s, 1, t\}$
- $t \mapsto \{s, 1, t\}$
- $a(s, 1) \mapsto \{a(s, 1)\}$
- $a(s, 2) \mapsto \{a(s, 1)\}$
- $b(1, t) \mapsto \{b(1, t)\}$
- $b(2, t) \mapsto \{b(1, t)\}$

However, note that when doing this we essentially leave the input state space relatively large and generate a lot of candidate partial maps. This approach essentially relies on the remaining steps of the algorithm to filter through all the possible mappings to find a query homomorphism that works for the entire graph. While this does correctly determine query homomorphism, it is far from efficient. In particular, due to the absence of any rules for vertex mapping, we are allowing every vertex in the input graph to be mapped to every vertex in the graph we are testing for containment. Moreover, one of the next steps involves computing a Cartesian product of these partial mappings, further increasing the size of the search space. Consequently, it would be beneficial to make our rules for determining partial mappings as strict as possible to reduce the size of the search space. In order to do this, we will take into account more of the graph when determining if a partial mapping exists, leading to the following rules.

For vertices v and v' to test if $v \mapsto v'$:

1) If v is a source vertex then v' has to be a source vertex.

2) If v is a target vertex then v' has to be a target vertex.

3) If an edge with label l leaves v, an edge with label l has to leave v'.

4) If an edge with label l enters v, an edge with label l has to enter v'.

For edges $e = (u, v, l)$ and $e' = (u', v', l)$ to test if $e \mapsto e'$:

1) Edge labels match: $l = l'$.

2) A partial mapping exists between the source vertices: $u \mapsto u'$.

3) A partial mapping exists between the target vertices: $v \mapsto v'$.

These new rules result in a much smaller search space for the next steps of the algorithm without being overly expensive to compute. Also note that the extra rules are essentially just implicit conditions that were derived from the definition of a query homomorphism. Naturally, edges can only be mapped if their end points are compatible. Consequently, vertices can only be mapped if they have edges with the same labels entering and leaving them. Note that we cannot assume anything about the number of edges with a specific label entering or leaving. It is possible that some of these edges with identical labels will collapse by both mapping to the same target edge. However, note that it is not possible for all edges with a specific label to disappear.

Using these new rules we can now again determine the partial maps for our running example *CPQ*s q and q' from Figure 3.3. Note that this time each mapping only has one valid target. Though note that it is theoretically still possible for there to be many possible valid targets for a mapping. Finally, note that if any of these mappings have an empty right hand side, then there is never a query homomorphism between the input graphs. In this case the answer can be returned immediately and the computation of an incidence graph and tree decomposition can also be skipped.

- $s \mapsto \{s\}$
- $1 \mapsto \{1\}$

- $2 \mapsto \{2\}$
- $t \mapsto \{t\}$
- $a(s,1) \mapsto \{a(s,1)\}$
- $a(s,2) \mapsto \{a(s,1)\}$
- $b(1,t) \mapsto \{b(1,t)\}$
- $b(2,t) \mapsto \{b(1,t)\}$

After determining these mappings we need to extend the tree decomposition with this information. Recall that each bag of the tree decomposition is made up of at most 3 elements that are also used as the left hand side of our partial mappings. Essentially, our goal is to compute a partial map for each tree decomposition bag as a whole. Note that this comes down to computing the Cartesian product of the partial mappings for each element of the tree decomposition bag. For a tree decomposition bag X, we would need to compute Equation 3.4.

$$X = \{x_1, \ldots, x_n\} \mapsto (x_1 \mapsto \{y_1, \ldots, y_{i_1}\}) \times \cdots \times (x_n \mapsto \{y_1, \ldots, y_{i_n}\}) \tag{3.4}$$

Note that as mentioned previously this set can quickly grow in size if the number of input partial maps is not kept low. However, it is also worth noting that our bags are always of size 2 or 3, so the impact of the Cartesian product is limited. Knowing how to compute the partial mapping for a tree decomposition bag we can now extend the tree decomposition by adding the partial mappings. This augmented tree decomposition with the partial mapping targets highlighted in red is shown in Figure 3.6.

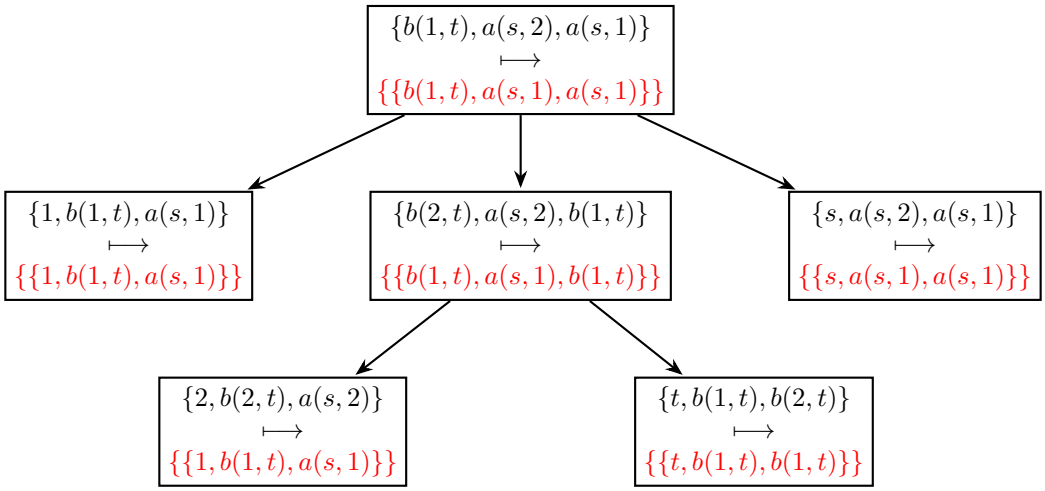


Figure 3.6: The example graph tree decomposition with partial maps.

Note that each bag only has a single target since each of the original maps also only had one target. In practice it is likely that more targets are present. However, this would needlessly complicate the example. Finally, it is worth mentioning in advance that some of the computed candidate partial maps are inherently inconsistent and as a result invalid. We will cover how to remove these invalid partial maps in Section 3.1.3.4.

3.1.3.4 Dependent Variables in Partial Mappings

After computing the candidate partial maps, the next step is to extend these maps with dependent variables. Unlike the independent variables we worked with so far that were directly derived from a vertex or an edge, dependent variables are derived from independent variables. Specifically, note that when we construct a partial mapping for an edge we are technically not only mapping the edge, but also implicitly the endpoint vertices of this edge. More formally we have Equation 3.5:

$$(l(x_1, x_2) \mapsto l(y_1, y_2)) \implies (x_1 \mapsto y_1) \wedge (x_2 \mapsto y_2) \tag{3.5}$$

For our next step we will be making these implicit mappings explicit. We will do this by extending both the left hand side of the partial mapping at each tree decomposition node and all targets at the same time. It is important to keep in mind that the data in a tree decomposition node are essentially ordered sets representing multiple partial mappings. This means that if the i-th element of the tree decomposition bag is an edge, then

the i-th element in all the targets stored at that node is also an edge. The process of computing dependent variables then works as follows. First we check each element of the node bag in order. If the variable is a vertex then we do nothing. However, if the variable is an edge, we check if one of the end points is not already an existing independent variable in the bag. If one or both endpoint vertices are not already represented by an existing independent variable, then we extend the node bag with this vertex and at the same time do the same with the edges in the same positions in each stored target. To give an example, suppose we have the candidate partial mapping shown in Equation 3.6 that has two targets.

$$\{v_1, l(v_1, v_2)\} \mapsto \{\{v_3, l(v_3, v_4)\}, \{v_5, l(v_5, v_6)\}\} \tag{3.6}$$

Observe that there are two potential dependent variables in this mapping in the form of the endpoints v_1 and v_2 of the edge $l(v_1, v_2)$. However, notice that v_1 is already present as an independent variable, so it does not generate a dependent variable. The other endpoint v_3 does not occur already and thus will generate a dependent variable in all three sets for the implied mapping $v_2 \mapsto \{v_4, v_6\}$, this results in the partial mapping shown in Equation 3.7.

$$\{v_1, l(v_1, v_2), v_2\} \mapsto \{\{v_3, l(v_3, v_4), v_4\}, \{v_5, l(v_5, v_6), v_6\}\} \tag{3.7}$$

Note that the order matters, the newly added dependent variables are added at the end of each set and consequently are at the same index in each set. A simple algorithm to extend partial mappings with dependent variables is shown in Algorithm 2.

Algorithm 2 Computing dependent variables from independent variables.

```
1: procedure ComputeDependentVariables((X = {x_1, ..., x_n}) ↦ (𝒴 = {Y_1, ..., Y_m}))
2:     for i = 1 to n do
3:         if x_i is an edge then
4:             if x_i.u ∉ X then        ▷ Check if the first vertex of the edge is already an independent variable.
5:                 X ← X ∪ {x_i.u}
6:                 for Y ∈ 𝒴 do
7:                     y_i ∈ Y                                  ▷ Get the i-th element of Y.
8:                     Y ← Y ∪ {y_i.u}
9:                 end for
10:            end if
11:            if x_i.v ∉ X then     ▷ Check if the second vertex of the edge is already an independent variable.
12:                X ← X ∪ {x_i.v}
13:                for Y ∈ 𝒴 do
14:                    y_i ∈ Y
15:                    Y ← Y ∪ {y_i.v}
16:                end for
17:            end if
18:        end if
19:    end for
20: end procedure
```

Recall from section 3.1.3.3 that some of the computed candidate partial maps were inherently inconsistent and invalid. The reason for this has to do with dependent variables. Consider the partial mapping shown in Equation 3.8.

$$\{v_1, l(v_1, v_2)\} \mapsto \{v_3, l(v_4, v_5)\} \tag{3.8}$$

Notice that this mapping contains two implicit mappings in the form of $v_1 \mapsto v_4$ and $v_2 \mapsto v_5$. However, there was already an existing mapping for v_1, namely $v_1 \mapsto v_3$. This would imply that to use this mapping, v_1 needs to be mapped to v_3 and v_4 at the same time. Naturally this is impossible, the mapping contradicts itself. Another variant of this issue arises when two edges both generate different dependent variable mappings. Thus before moving on to the final stage of the containment checking algorithm, we need to make sure to filter out these invalid mappings. Notice that this can easily be done at various places in the algorithm, the easiest being to perform a filtering operation directly after the Cartesian products are computed. However, modifying the Cartesian product operation itself or performing the filtering when computing dependent variables are also valid solutions. The reference implementation for this thesis uses both approaches and this algorithm will be

discussed in Section 3.1.3.6. Also note again that we can immediately conclude that there is no containment relation if any of the mappings end up with an empty right hand side after filtering.

Finally, we extend the tree decomposition with partial maps for our example *CPQ* with dependent variables. This augmented tree decomposition is shown in Figure 3.7 with the dependent variables highlighted in red.

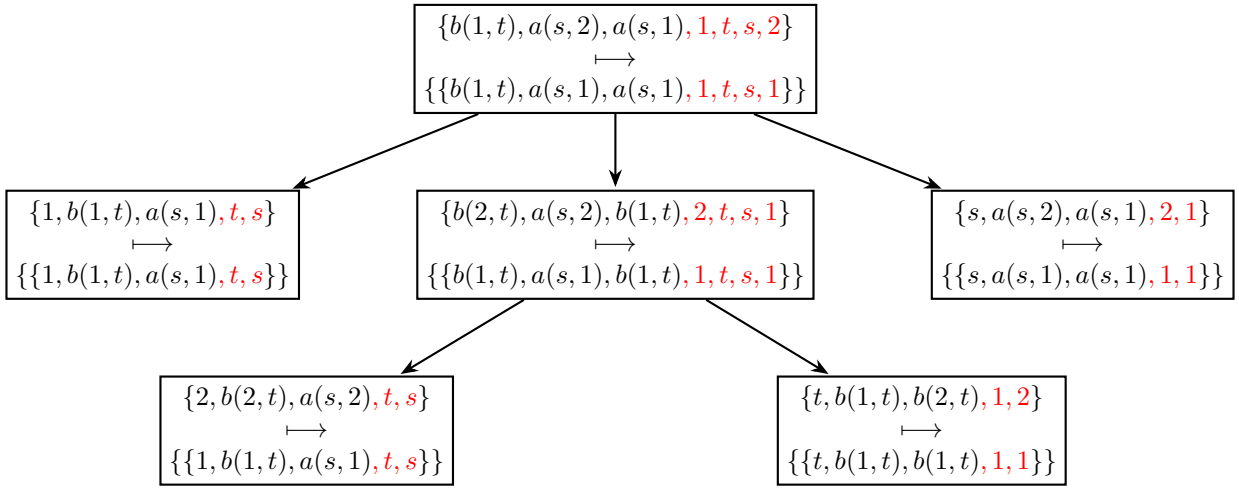


Figure 3.7: The example graph tree decomposition with dependent variables.

3.1.3.5 Verifying Containment

The final step of the algorithm involves processing the augmented tree decomposition bottom up and computing the natural left semijoin of each node with its children. Concluding whether the test query is contained in the other query is then a matter of checking if the mapping at the root node of the tree is empty. Before showing how this works in detail, we first need to introduce the notion of a natural left semijoin '$\ltimes$'. Essentially, this operation joins two tables on columns with matching names and only keeps the columns from the left table in the result. As we will see later, this effectively results in the left table being filtered by the right table.

We will give an example using the tables in Table 3.1, collectively these tables have three unique columns **A**, **B** and **C**. Note that we can always classify the involved columns as one of three categories, columns that are only in the left table, columns that are only in the right table, and columns that are in both tables. Columns exclusive to the first table are only copied and otherwise not relevant and columns exclusive to the right table are completely dropped, thus the shared columns are most interesting. Rows in the input tables that share the same value for all shared columns are used to populate the output table. Note that since the columns exclusive to the right table are dropped, this effectively filters the rows in the left table. For completeness, the output rows in Table 3.1c are produced by the semijoins $\{1,2\} \ltimes \{2,3\} = \{1,2\}$ and $\{3,4\} \ltimes \{4,1\} = \{3,4\}$.

Table 3.1: Example of a natural left semijoin between two tables.

(a) The left input table.

A	**B**
1	2
2	3
3	4

$\ltimes$

(b) The right input table.

B	**C**
2	3
1	2
4	1

$=$

(c) The output table.

A	**B**
1	2
3	4

Having introduced the natural left semijoin we will now show how to process the augmented tree decomposition. Given a tree decomposition node, we will treat the left hand side of the partial mapping as defining the names of the columns and all targets are rows in the table. For example, using the root node of the tree decomposition in Figure 3.7 we obtain the relation in Table 3.2.

Table 3.2: Partial mapping as a table.

b(1, t)	**a(s, 2)**	**a(s, 1)**	**1**	**t**	**s**	**2**
$b(1,t)$	$a(s,1)$	$a(s,1)$	1	t	s	1

The next step is to perform a natural left semijoin between the relation for each node with the relations of its child nodes. For this procedure it is important to process children before their parent is processed. Otherwise not all the information required to properly process a parent node will be available. For example, for the center node of the tree decomposition in Figure 3.7 we compute Equation 3.9. Keep in mind that these sets are rows from relations with column names, the attribute name data is not shown in this equation.

$$\begin{aligned}(\{\{b(2,t),a(s,2),b(1,t),2,t,s,1\}\} \ltimes \{\{2,b(2,t),a(s,2),t,s\}\}) \ltimes \{\{t,b(1,t),b(2,t),1,2\}\} \\ = \{\{b(2,t),a(s,2),b(1,t),2,t,s,1\}\}\end{aligned} \tag{3.9}$$

After computing these semijoins for each node in the tree decomposition all we need to do is check if the relation at the root node is empty. If there are no mapping targets remaining at the root node then there is no containment relation between the two conjunctive queries. Conversely, if the root is not empty, then the tested query is contained in the other query, meaning the output of the tested query is a subset of the other query. For our running example we were expecting that there was a containment relation and looking carefully at Figure 3.7 we can see that root node will not turn empty. Therefore, q is contained in q' meaning q is query homomorphic to q'. Furthermore, note that if during this semijoin process the relation at any intermediate node ends up empty, then the root will always end up empty as well. This means that processing can be stopped early in these cases. Finally, it is worth noting that there are some potential optimisations to the presented approach. These optimisations will be discussed in more detail as future work in Section 6.1.1 except for one major optimisation that was actually implemented, this optimisation will instead be covered in Section 3.1.3.6.

3.1.3.6 Improving Performance

By far the most computationally expensive part of the containment procedure is the Cartesian product step and the post processing related to it. As such, it makes sense to focus on this part of the procedure for optimisations. In this section we will present an approach that combines the Cartesian product, filtering and dependent variables step into a single algorithm. This algorithm is given in Algorithm 3 and consists of two main parts. The first part computes a new left hand side for the partial mapping with dependent variables and derives some additional information required for the second part. The second part then actually computes the right hand side for all partial mappings using the information computed during the first part. The main information that is passed between these two steps is the location of vertex attributes for edges in the mapping sets, which is used to filter out inconsistent and thus invalid mappings and to determine which dependent variables to generate.

Note that performance wise this algorithm still needs to compute a number of Cartesian products, so the runtime trivially scales with product of the input set sizes. This gives the algorithm a runtime of $\mathcal{O}(|S_1| \times \cdots \times |S_n|)$, for input sets $S_1, \ldots, S_n$.

Algorithm 3 Computing partial mappings from individual attribute mappings.

```
1:  procedure ExpandPartialMap(M, 𝒦)        ▷ Partial map M and a map of known attribute mappings 𝒦.
2:      𝒮 ← ∅                                ▷ Set of sets to compute the Cartesian product of.
3:      L ← ∅                                ▷ New left hand side of the mapping with dependent variables.
4:      for a ∈ M.left do                    ▷ For all dependent variables.
5:          if a is edge then
6:              s ← index(L, a.src)          ▷ The index of a.src in L or -1.
7:              t ← index(L, a.trg)
8:              if a.src = a.trg then        ▷ Prevent loops from generating the same dependent variable twice.
9:                  t ← −2
10:             end if
11:             a.refs ← (s, t)              ▷ Store references to endpoint attributes.
12:             𝒮 ← 𝒮 ∪ {𝒦[a]}               ▷ Known mapping targets for a.
13:             L ← L ∪ {a} ∪ {a.src | s = −1} ∪ {a.trg | t = −1}   ▷ Add endpoints unless already independent.
14:         else if a ∉ L then               ▷ Directly add vertices unless already present.
15:             L ← L ∪ {a}
16:             𝒮 ← 𝒮 ∪ {𝒦[a]}
17:         end if
18:     end for
19:     M.left ← L                           ▷ New left hand side with dependent variables.
```

```
20:     s ← Π_{A∈S} |A|
21:     P ← {P_1, ..., P_s}                                  ▷ Cartesian product output sets.
22:     for S ∈ S do                                         ▷ Build the product set by set.
23:         s ← s / |S|
24:         i ← 1
25:         for o ← 1 to |P| do
26:             if P_i ≠ nil then                            ▷ If nil this candidate was already deemed invalid.
27:                 a ← S[o mod |S|]
28:                 for k ← i to i + s do
29:                     if a is edge then
30:                         (r_s, r_t) ← a.refs
31:                         if (r_s ≥ 0 ∧ P_i[r_s] ≠ a.src) ∨ (r_t ≥ 0 ∧ P_i[r_t] ≠ a.trg) ∨ (r_t = −2 ∧ a.src ≠ a.trg) then
32:                             P_k ← nil                    ▷ These candidates are invalid.
33:                         else
34:                             P_k ← P_k ∪ {a} ∪ {a.src | r_s = −1} ∪ {a.trg | r_t = −1}
35:                         end if
36:                     else
37:                         P_k ← P_k ∪ {a}
38:                     end if
39:                 end for
40:             end if
41:             i ← i + s
42:         end for
43:     end for
44:     M.right ← {a | a ∈ P ∧ a ≠ nil}                      ▷ Filter out candidates that were deemed invalid.
45: end procedure
```

3.2 Computing *CPQ* Cores

Now that we have introduced the query homomorphism test we needed, we can proceed to show how the core of a *CPQ* can be computed. The work that will be presented in this section is based on several unpublished prior research projects [47, 39, 26]. Recall that we are interested in *CPQ* cores because their evaluation result on a graph is the same as the evaluation result of the original query graph. As such, it makes sense to use the core of a query instead of the original query wherever possible to reduce the amount of processing and storage required for various tasks in the index. Moreover, we can compute a *CPQ* core for any *CPQ* and *CPQ*s with equivalent cores are query homomorphic. Note that this is an extremely important property for our index key to have. Naively, we could decide to use *CPQ*s as index keys directly and accept the additional storage overhead. However, note that while the number of *CPQ* cores of a given diameter in a graph is finite, the number of *CPQ*s with a given diameter is not, leading to the issue that was described in Section 1.1. The reason the number of *CPQ*s of a given diameter is infinite follows from the fact that you can always duplicate existing structure in a *CPQ* to make a new *CPQ* that matches the same paths.

To compute a *CPQ* core we first start by formalising its definition, which is done by modifying the definition of a regular graph core as introduced in Section 2.2.7 to use a query homomorphism as defined in Section 3.1.1 instead of a regular homomorphism. This establishes a *CPQ* core as the smallest subgraph of the query graph of a *CPQ* that is query homomorphically equivalent to the query graph it is a subgraph of. This modification essentially ensure that the definition of a *CPQ* core inherits the notion of a fixed source and target vertex from the definition of a query homomorphism. Note that this definition guarantees that the source and target vertex of the query core match the same parts of the original graph as the original query, since the source and target vertex determine which vertex pairs are returned in the evaluation. We will also formally establish this fact.

Theorem 5. *Let $q, q' \in CPQ$ and q' is a query core of q, then for a graph $\mathcal{G}$ it holds that $[\![q]\!]_{\mathcal{G}} = [\![q']\!]_{\mathcal{G}}$.*

Proof. In order to prove that the evaluation result of the core is equal to the evaluation result of the original query, that is $[\![q]\!]_{\mathcal{G}} = [\![q']\!]_{\mathcal{G}}$, we will separately prove a containment relation in both directions. More precisely, we will show that $[\![q]\!]_{\mathcal{G}} \supseteq [\![q']\!]_{\mathcal{G}}$ and $[\![q]\!]_{\mathcal{G}} \subseteq [\![q']\!]_{\mathcal{G}}$, together implying that $[\![q]\!]_{\mathcal{G}} = [\![q']\!]_{\mathcal{G}}$.

We start with $[\![q]\!]_{\mathcal{G}} \supseteq [\![q']\!]_{\mathcal{G}}$. Note that q' being a core of q means that q and q' are query homomorphically equivalent. This means that a query homomorphism f exists from $\mathcal{G}_q$ to $\mathcal{G}_{q'}$, which implies that any source

target node pair matched by q can also be matched by q'. If not, f would not be a query homomorphism, thus leading to a contradiction. Essentially we are combining the regular homomorphism that exists from the query to the graph it is being evaluated on and the query homomorphism between the query and its core. Since a query homomorphism also exists from $\mathcal{G}_{q'}$ to $\mathcal{G}_q$ the proof that $[\![q]\!]_{\mathcal{G}} \subseteq [\![q']\!]_{\mathcal{G}}$ is similar, we again combine the query homomorphism from the core to the query and the regular homomorphism from the core to the graph. Essentially we just perform the same steps in the opposite direction. □

Next we will show a method to compute the query core of an input *CPQ* query graph. First recall that a query core is a subgraph of the original query graph. This means that a set of edges exists that can be removed from the original query graph to form the query core. We can construct a simple algorithm based on this idea that simply tries to remove each edge while checking if there is still a query homomorphism from the original query graph to the result. Note that this test will be performed by the containment testing procedure introduced in Section 3.1. When this test confirms that a query homomorphism still exists the edge is permanently removed, otherwise the edge is part of the result. Note that removing edges from the graph may result in vertices ending up with no edges attached to them. These vertices obviously do not belong in the result and should thus be removed from the core graph before the algorithm concludes. It is worth noting in advance that this algorithm is not the most efficient method to compute a core, we will introduce a more efficient approach in Section 3.3. However, the method presented here is far more intuitive than the improved approach we will present in that section. The pseudocode for the entire algorithm can be found in Algorithm 4.

Algorithm 4 *CPQ* Core Computation

procedure ComputeQueryCore($\mathcal{G}_q = (\mathcal{V}_q, \mathcal{E}_q, \mathcal{L}, v_s, v_t)$) ▷ The query graph $\mathcal{G}_q$ of $q \in CPQ$
  $\mathcal{G}_{core} \leftarrow \mathcal{G}_q$
  for $e \in \mathcal{E}_{core}$ **do**
    $\mathcal{G}_{q'} \leftarrow (\mathcal{V}_{core}, \mathcal{E}_{core} \setminus e, \mathcal{L}, v_s, v_t)$
    if IsQueryHomomorphicTo($\mathcal{G}_q, \mathcal{G}_{q'}$) = **true then**
      $\mathcal{G}_{core} \leftarrow \mathcal{G}_{q'}$
    end if
  end for
  for $v \in \mathcal{V}_{core}$ **do**
    if $\nexists_{e \in \mathcal{E}_{core}}[v \in e]$ **then**
      $\mathcal{G}_{core} \leftarrow (\mathcal{V}_{core} \setminus v, \mathcal{E}_{core}, \mathcal{L}, v_s, v_t)$
    end if
  end for
  return $\mathcal{G}_{core}$
end procedure

Having given the algorithm, we will now formally prove its correctness.

Theorem 6. *Given a* CPQ *query graph* $\mathcal{G}_q$*, the* CPQ *core* $\mathcal{G}_{core}$ *of* $\mathcal{G}_q$ *can be computed by Algorithm 4.*

Proof. First it is worth noting that we will assume the correctness of the `IsQueryHomomorphic` subroutine. The correctness of the core idea for this subroutine was already established in the paper it is based on [17] and an alternative query homomorphism test could also be used without affecting the proposed algorithm.

Note that showing the correctness of the algorithm as a whole comes down to establishing two properties:

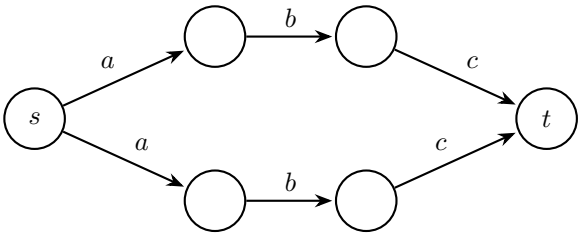
1) For the first property, we require that the computed core is actually a core and thus no smaller homomorphically equivalent subgraph exists. Note that the algorithm itself trivially establishes this. If any edge still remained that could be removed, then it would already have been removed by the algorithm. It then remains to establish that no other order of removing edges would have yielded a smaller core. Note that this follows from the fact that all the computed core candidates are query homomorphically equivalent, save for the vertices without any attached edges we remove at the end. However, note that these obsolete vertices would only interfere with a homomorphism test from the core to the input graph, which is not a test we need to perform. Note that the main connected component that makes up the core is always a subgraph of the input graph at any stage and thus always query homomorphic to the input graph. Therefore, given that these graphs are always homomorphically equivalent the removal order is not relevant to the end result.

2) The second property we need to establish is that the computed core actually represents a valid *CPQ*. Note from the query graph construction shown in Section 2.2.3 that an edge only appears in two ways in the graph, as an edge in a simple path of edges or as an edge in one of the branches for a conjunction.

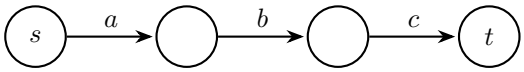
These two cases can be characterised by the *CPQ* grammar as $q \circ l$ and $q \cap l$, where $q \in CPQ$ and $l \in \mathcal{L}$. This also clearly shows that dropping the edge with the l label in these two cases still leaves a valid *CPQ* provided we also remove the adjacent operator. Essentially, we effectively just shorten the path for the join case and for the conjunction case one of the branches is removed leaving the remaining branch as the only path. Even if these subgraphs appear as part of a larger *CPQ* query graph, the edge removal still has the same effect. Thus it remains to show that this procedure will never result in a disconnected query graph. Note that the conjunction case will clearly never result in a disconnected graph. However, this is not immediately clear for the join case. For the join case note that there is a query homomorphism between the original graph and the new core graph, thus either there exists some other path still keeping the graph connected or a contradiction arises where there was no query homomorphism, since a connected graph cannot be homomorphic to a disconnected graph.

Therefore, given that both properties hold, the proposed algorithm correctly computes *CPQ* cores. □

To conclude this section, we will give some examples of *CPQ*s and their cores. These examples are given in Figures 3.8, 3.9, 3.10 and 3.11, and include three simple cases and one more complex core. In general, note that many cores are of the trivial variety where some subgraph is exactly duplicated a number of times. However, more complex cores such as Figure 3.9 also exist. When carefully inspecting this core example it becomes clear that all paths involving the longer removed path can be simulated using the shorter paths that remain.

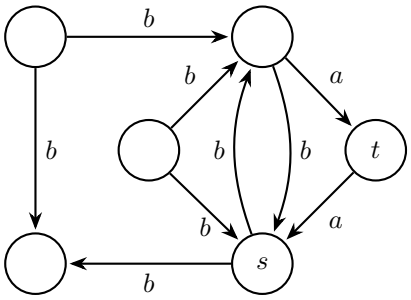


(a) Graph for $q = (a \circ b \circ c) \cap (a \circ b \circ c)$.

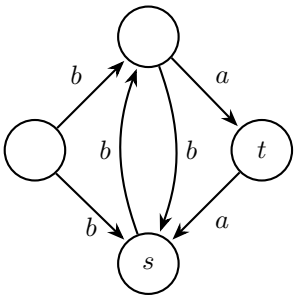


(b) The core of q is $a \circ b \circ c$.

Figure 3.8: Simple *CPQ* core example where the same path is present twice in the input.

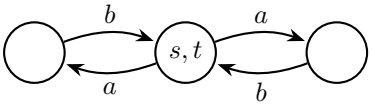


(a) Graph for $q = a^- \cap ((b \cap b^- \cap (b^- \circ b) \cap (b \circ b^- \circ b)) \circ a)$.

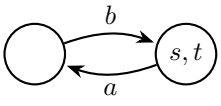


(b) The core of q is $a^- \cap ((b \cap b^- \cap (b^- \circ b)) \circ a)$.

Figure 3.9: Complex *CPQ* core example where a longer path can be simulated using multiple shorter paths.

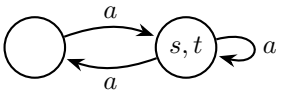


(a) Graph for $q = (a \circ b) \cap (a \circ b) \cap id$.



(b) The core of q is $(a \circ b) \cap id$.

Figure 3.10: Simple *CPQ* core example where the same cycle is present twice in the input.



(a) Graph for $q = (a \circ b) \cap a \cap id$.

(b) The core of q is $a \cap id$.

Figure 3.11: *CPQ* core example where the larger cycle can be simulated by going through the self loop twice.

3.3 Efficiently Computing *CPQ* Cores

The algorithm for *CPQ* core computation presented in Algorithm 4 in Section 3.2 sequentially performs a large number of query homomorphism tests from the same graph to subgraphs. In fact, the number of tests scales linearly with the number of edges in the query graph, as an attempt is made to remove each edge. Considering that a lot of the data computed for a query homomorphism test does not change significantly when removing a single edge, this approach seems inefficient. Moreover, given the nature of the homomorphism test, it should be possible to extract the exact homomorphism mapping that was found. The reason the original authors did not consider this direction is likely due to performance concerns and because tracking the required data is irrelevant for a simple test. However, for our use case it does make sense to investigate this direction, as it would mean we can switch to a single test.

In this section we will introduce a novel algorithm for directly computing *CPQ* cores that uses many of the same ideas as the homomorphism test originally introduced in Section 3.1.3. It is worth noting in advance that while this algorithm will be designed to compute *CPQ* cores, extending it to work for vertex labels, regular graph cores, or even novel augmented homomorphism definitions is trivial. One part from the original algorithm we will be reusing without any changes is the mapping step where components from one of the input graphs are mapped to potential target components in the other input graph. This procedure was originally discussed in Section 3.1.3.3 and effectively enforces what components can map to each other. As such, adding or removing restrictions similar to what we did to support a query homomorphism is relatively straightforward.

3.3.1 General Approach

In this section we will introduce the main idea behind our new algorithm. Notice that the partial mappings that are computed during a homomorphism test can be combined to form a mapping for the entire graph given some extra bookkeeping. Based on this idea we will essentially run a homomorphism test from the input graph to itself. Trivially, following the original test we would always discover a query homomorphism this way. However, note that the data technically contains information on all possible mappings, and not just the identity mapping. Essentially what we will do is process the tree decomposition bottom up like in the original algorithm while also keeping track of complete graph mappings. Since keeping track of all possible mappings is potentially very resource intensive, we will also show how to filter this pool of candidates continuously to only keep relevant candidates in memory. Note that this means that compared to the original algorithm, we are essentially replacing the natural left semijoins with modified regular joins, and we will also be interpreting the final relations at the root node differently. During this section, we will make use of the same running example graph originally used for the homomorphism test in Section 3.1.3, this graph is also shown again in Figure 3.12.

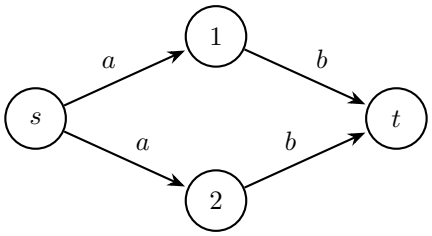


Figure 3.12: Running example *CPQ*s for the *CPQ* core algorithm.

Note that this time we will be mapping this graph onto itself. This means that while the tree decomposition will be identical to the one computed previously, the partial mapping targets will be different. Notably, each map will have multiple targets this time. In addition, we will sort the attributes for the relation at each node such that it becomes possible to execute sort merge joins. The exact sorting order used does not matter as long as it is consistent, the reference implementation simply assigns a unique identifier to each vertex and edge. Note that it does not matter if the sorting algorithm is stable or not as all the items to sort will be distinct. Following these minor modifications the new tree decomposition with all partial maps fully computed and all semijoins computed is shown in Figure 3.13. For each tree decomposition node the attributes or column names are listed along the top of the cell. The possible target mapping candidates are listed on separate rows with their mapping targets in the order that matches the attributes they map from. In particular, note that there are two vertices in the example graph that have an identical neighbourhood, namely the vertices labelled with 1 and 2. As a result, all the candidates shown in the tree decomposition are varying maps for these vertices.

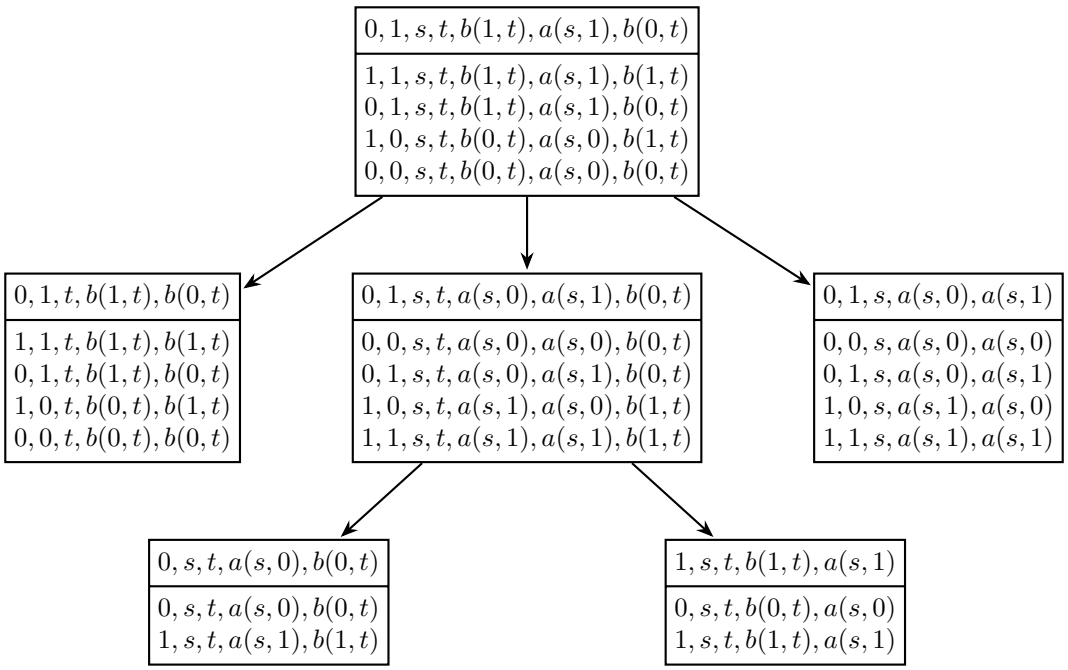


Figure 3.13: The example graph tree decomposition with partial maps to itself.

3.3.2 New Semijoins

In this section we will introduce the replacement for the semijoins. Recall that for a semijoin we distinguished 3 categories of columns in Section 3.1.3.5, columns only in the left table, columns in both tables, and columns only in the right table. By using a semijoin the columns only in the right table are essentially completely ignored. However, these columns do contain important mapping information required to construct a mapping for the graph as a whole. Notice though that these attribute mappings are never going to be seen again. If during a semijoin some attribute is not present in the parent node, then by definition of a tree decomposition no tree decomposition nodes past the parent will contain this attribute, since each attribute induces a connected subtree. This is true even for vertex attributes that are hidden inside of an edge attribute, as these were already extracted by the algorithm that computes dependent variables covered in Section 3.1.3.4 and Section 3.1.3.6. This means that while we need to keep track of these attribute mappings, we can actively filter them without affecting correctness and always only keep the smallest candidates.

Note that essentially we want to compute regular joins. However, since we are not joining on some unique key such a join could result in a lot of rows being generated. For this reason we will instead opt to track the mappings that are no longer relevant as metadata for each row. We will also start referring to these mappings as free mappings, given that they are no longer relevant to the main computation. To give a more concrete example we show a semijoin between two relations tracking free mappings in Table 3.3. In particular, note that each joining row generates its own set of free mappings, so rows that join more than once receive more than one possible set of free mappings that complements it. Since free variables are no longer relevant to the computation, any of these options is a correct extension to form the complete graph mapping. This is the core idea behind actively filtering them later on.

Table 3.3: Example semijoin between two tables with free output mappings.

(a) The left input table.

A	**B**
1	2
2	3
3	4

$\ltimes$

(b) The right input table.

B	**C**
2	3
2	2
4	1

$=$

(c) The output table.

A	**B**	**Free**
1	2	$\{\{C \mapsto 3\}, \{C \mapsto 2\}\}$
3	4	$\{\{C \mapsto 1\}\}$

However, also note that this is just a simplified example. Naturally, if the output tables of a semijoin now contain free mappings, then these free mappings can also appear in the input tables. In Table 3.4 the four main cases are shown for how to process free mappings during a semijoin. Note that essentially each set of mapping options represents multiple rows that would have been generated during a regular join. These sets are extended

when new free mappings are generated and existing sets in the left input table are just copied without any changes.

Table 3.4: Processing free mappings when joining relations.

(a) The left input table.

A	**B**	**Free**
1	2	$\{\{D \mapsto 4\}\}$
2	3	
3	4	$\{\{D \mapsto 4\}\}$
4	5	

$\ltimes$

(b) The right input table.

B	**C**	**Free**
2	3	
3	2	$\{\{E \mapsto 3\}\}$
4	1	$\{\{E \mapsto 3\}\}$
5	0	

$=$

(c) The output table.

A	**B**	**Free**
1	2	$\{\{D \mapsto 4\}\}, \{\{C \mapsto 3\}\}$
2	3	$\{\{E \mapsto 3, C \mapsto 2\}\}$
3	4	$\{\{D \mapsto 4\}\}, \{\{E \mapsto 3, C \mapsto 1\}\}$
4	5	

Note that these examples exclude any cases where the right table has an existing set of free mappings with more than one set. We will discuss this case separately as this case represents the most performance intensive part of the algorithm. Note that in the trivial case, such as the join on the second row in Table 3.4, we just append a newly generated free mapping to the mapping that was already present. Essentially, we already had a set of free variables required for a full mapping and now we have one extra mapping required which we just append. However, this is more complicated when one of these option sets has more than one option for mapping the same free attribute. For example, consider the set of sets of free attribute mappings in Equation 3.10. Also note that the extra outermost pair of brackets was also present in the previous examples, just not explicitly written down.

$$\{\{\{A \mapsto 3\}, \{A \mapsto 2\}\}, \{\{B \mapsto 1\}, \{B \mapsto 3\}\}\} \tag{3.10}$$

Essentially, this set represents four rows that would have been generated by a regular join. As such, when deriving the set of options to produce we essentially need to restore all these rows. Note that this means that this step is a Cartesian product of all the option sets. To give a more concrete example, assume that we are joining the row from Equation 3.10 with a different row that does not yet have any free mappings. Also assume that we are generating one new free mapping $C \mapsto 2$ during this join, then we obtain the result shown in Equation 3.11.

$$\{\{A \mapsto 3, B \mapsto 1, C \mapsto 2\}, \{A \mapsto 3, B \mapsto 3, C \mapsto 2\}, \{A \mapsto 2, B \mapsto 1, C \mapsto 2\}, \{A \mapsto 2, B \mapsto 3, C \mapsto 2\}\} \tag{3.11}$$

From this equation we can see that the original four rows have all effectively been restored completely, and each has been extended with the newly generated free variable. As previously mentioned, the next step is to filter this set to only keep relevant candidates. For clarity, the rows from Equation 3.11 have been tabulated in Table 3.5, together with the mappings of the row that have not been classified as free yet. Note that these other attributes are always the same for all entries in the table, as this data is all for the same row. A cost value has also been assigned to each row, which will be explained in more detail next.

Table 3.5: Complete candidate mapping rows with their associated cost.

Not Free			**Free**			
D	**E**	**F**	**A**	**B**	**C**	**Cost**
0	1	1	3	1	2	4
0	1	1	3	3	2	4
0	1	1	2	1	2	3
0	1	1	2	3	2	4

The cost value listed in Table 3.5 is computed by counting the number of distinct mapping targets across the entire row. Note that when we recombine non-free and free mappings like this, we are essentially computing possible mappings for a subgraph of the entire graph. However, while we cannot change the non-free attributes, we can freely decide which complementary set of free attributes to pick. In the example, the 3rd row has a set of free mapping options that reduces the number of distinct targets, so this will be the only one we keep during our computations. This means the final option set will be as shown in Equation 3.12 instead of what we originally computed in Equation 3.11.

$$\{\{A \mapsto 2, B \mapsto 1, C \mapsto 2\}\} \tag{3.12}$$

This filtering procedure keeps the size of the option sets small and also leads to smaller Cartesian product outputs at higher tree nodes. Note that it is possible for multiple options to have the same lowest cost. Unfortunately all lowest cost candidates need to be kept around in this case as the later addition of a new free mapping may make any of them a better choice. The complete new semijoin algorithm is shown in Algorithm 5. Note that this algorithm only saves the targets for free mappings. The source is useful when trying to understand the algorithm, but never actually used by the algorithm itself.

Algorithm 5 New semijoin algorithm for complete graph mappings.

```
1: procedure SEMIJOIN(P, C)                          ▷ A partial map P and one of its children C.
2:     for r_1 ∈ P.rows do
3:         m ← false                                  ▷ To track if this row joined with any row.
4:         O ← ∅                                      ▷ All sets of free mappings from all joins with the child partial map.
5:         for r_2 ∈ C.rows do
6:             T ← ∅                                  ▷ Free mappings for the current row join.
7:             i_1 ← 0
8:             i_2 ← 0
9:             while i_2 < |C.attr| do                ▷ Start of the sort merge join (pre-sorted).
10:                if a_1 < |P.attr| then
11:                    if P.attr[i_1] = C.attr[i_2] then   ▷ Matching attributes means targets need to match too.
12:                        if r_1[i_1] ≠ r_2[i_2] then
13:                            continue with the next iteration on line 5
14:                        end if
15:                        i_1 ← i_1 + 1
16:                        i_2 ← i_2 + 1
17:                    else if P.attr[i_1] > C.attr[i_2] then
18:                        T ← T ∪ {C.attr[i_2]}      ▷ We only save the target of the free mapping.
19:                        i_2 ← i_2 + 1
20:                    else
21:                        i_1 ← i_1 + 1
22:                    end if
23:                else
24:                    T ← T ∪ {C.attr[i_2]}
25:                end if
26:            end while
27:            m ← true
28:            if r_2.free ≠ ∅ then                   ▷ Extend existing free mapping sets.
29:                {F_1, ..., F_n} ← r_2.free
30:                O ← O ∪ (T × F_1 × ··· × F_n)
31:            else if T ≠ ∅ then                     ▷ We add the first free mapping set.
32:                O ← O ∪ {T}
33:            end if
34:            if O ≠ ∅ then                          ▷ Add the free mapping candidates to the options for the row.
35:                r_1.free ← r_1.free ∪ O
36:            end if
37:            if m = false then                      ▷ If no other rows joined with this row it does not remain in the parent.
38:                P.rows ← P.rows \ {r_1}
39:            end if
40:        end for
41:    end for
42: end procedure
```

As a quick note on correctness, note that we are not fundamentally changing the definition of the join and just tracking extra information. Also note that as discussed in this section, the extra information is essentially the result of a regular join stored in a more compact manner. Essentially this new semijoin definition is only a complex sequence of data transformations, each of which is reversible and thus not really interesting to prove. However, our filtering operation is an exception and does more than just transforming the data. Thus we will reiterate the foundation for its correctness in a short proof.

Theorem 7. *The filtering of candidate free mappings based on lowest cost is correct.*

Proof. Note that the correctness of this approach is derived directly from the fact that free mappings are not relevant to the remaining computation. However, let us assume that we had some set of candidates mappings with two candidates where one had a strictly lower cost than the other, meaning the higher cost candidate was filtered. Now also assume that in fact, using the candidate with a higher cost would have led to an overall graph mapping with less distinct targets. We know that the non free mappings for the two candidates are identical, so these are not relevant to the difference. Thus, the difference has to be in the free mappings. However, this is contradictory as the cost of the chosen mapping was computed to be lower and the non free mappings contribute the same cost to all rows. Alternatively, there would need to be some map higher up in the tree where a free mapping was repeated, but note that this conflicts directly with the definition of a free mapping. Thus by contradiction we can safely select only the lowest cost candidates. □

Finally, the runtime of this algorithm is obviously in the worst case no better than just computing the join of both relations, which has a quadratic runtime in the number of rows. However, the assumption is that the constant filtering will prevent this from happing most of the time. For the worst case scenario it essentially has to be possible for every vertex to map to every other vertex and every edge to map to every other edge. Particularly with the definition of a query homomorphism, we usually will not get close to this situation due to the special nature of the source and target vertex and the existence of different edge labels.

3.3.3 Computing the Core

Having introduced the new semijoin algorithm this means we can now reprocess the tree decomposition using this algorithm. Doing this results in the updated tree decomposition shown in Figure 3.14, where the free mappings are listed at the end of each row.

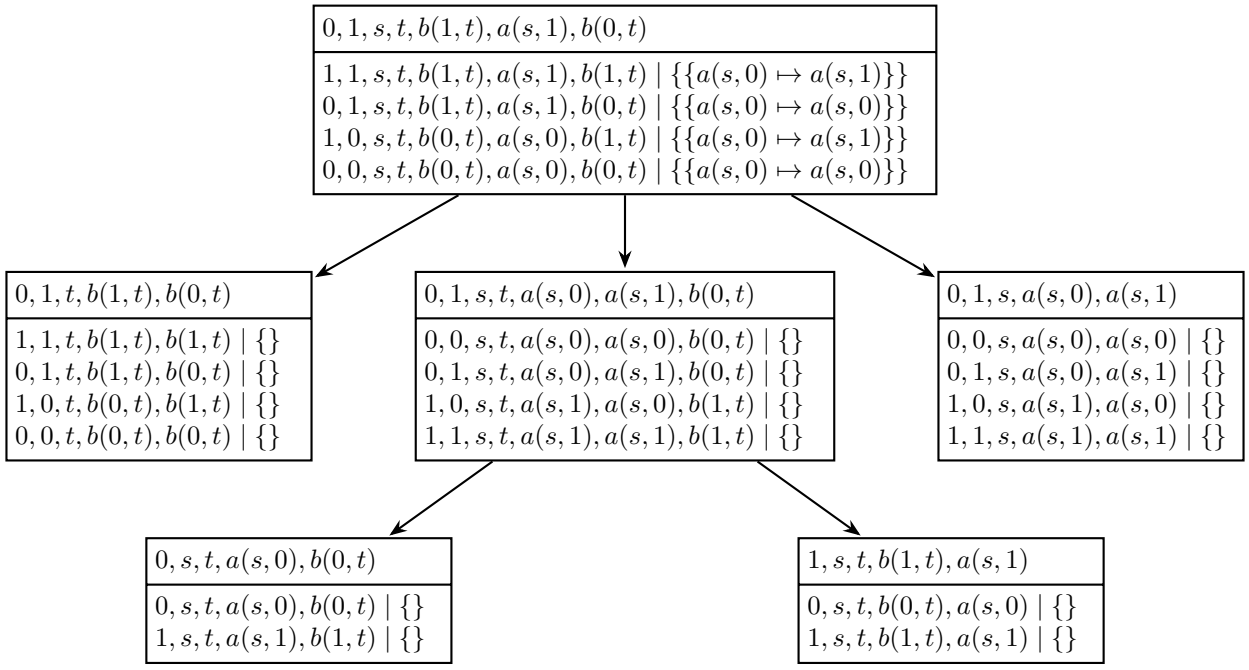


Figure 3.14: The example graph tree decomposition for core computation.

Note that due to the simplicity of the example only the root of the tree receives some mapping options. However, this does highlight the end step of the algorithm. After processing all semijoins we will end up with a number of rows at the root node. Each of these rows represents a complete mapping for the entire graph. However, these mappings do not necessarily all have the same number of distinct targets. Thus, we again need to compute a cost for each row and then pick the lowest cost row. Note that each child node of the root node contributes one set with sets of free mappings to choose from. Also note that different child notes will not contribute free mappings for the same attribute. Thus if a child node contributes a set with options, exactly one of the mapping sets has to be chosen. Note again that this is essentially a Cartesian product of the sets contributed by the child nodes. Essentially the general procedure greatly resembles the one used when joining option sets during semijoin computation. To compute the cost of a row we will again use the number of distinct targets in the lowest cost total mapping for each row. Note that since the example in Figure 3.14 has only one option set with

only one candidate there is only one candidate per row, but in practice rows can have more. The cost for each of the final rows with its best selection of free mappings is shown in Table 3.6.

Table 3.6: Rows at the root relation with their cost.

Not Free	**Free**	**Cost**
$1, 1, s, t, b(1,t), a(s,1), b(1,t)$	$a(s,0) \mapsto a(s,1)$	5
$0, 1, s, t, b(1,t), a(s,1), b(0,t)$	$a(s,0) \mapsto a(s,0)$	8
$1, 0, s, t, b(0,t), a(s,0), b(1,t)$	$a(s,0) \mapsto a(s,1)$	8
$0, 0, s, t, b(0,t), a(s,0), b(0,t)$	$a(s,0) \mapsto a(s,0)$	5

Note that this means that the example graph has two possible final mappings that are both equally good, both only using 5 distinct graph components. This is unsurprising as in terms of subgraphs either the vertex labelled with 1 or the vertex labelled with 2 from Figure 3.12 could be chosen for the core. For the relation at the root, any lowest cost option can be chosen as the complete graph mapping. Note though that at this point we still only have a query homomorphism mapping and not a core. The core can easily be derived from the original graph by removing all graph components that do not appear as a target in the final query homomorphism mapping.

3.4 Canonically Representing Cores

Now that we know how to compute the core of a *CPQ*, the next step is to convert this core to a form that is suitable for use in a database index. Note that we need to be able to quickly compare two cores for equivalence if we want to be able to use them as database keys. As a result, we want to compute some unique representation of a core, such that we can simply check this representation for equality. This representation is called a canonical form. Unfortunately, this means that our current representation of *CPQ* cores as graphs is completely unsuitable. Testing if two graphs are equal effectively comes down to testing for isomorphic equivalence. While it is still unknown in general how hard the graph isomorphism problem is exactly [54], it is considered unlikely that the problem is NP-complete, as this would collapse the polynomial-time hierarchy [29]. For more than three decades the fastest proven running time for graph isomorphism was $e^{\mathcal{O}(\sqrt{n \log n})}$ by Babai et al. [5], where n denotes the number of vertices in the graph. However, recently in 2015 László Babai managed to improve upon this bound and showed that the problem can be solved in quasipolynomial time $e^{(\log n)^{\mathcal{O}(1)}}$ [4], where n is again the number of vertices. Unfortunately, neither of these bounds are very promising for an index where speed is the primary concern. Given the high cost of isomorphism tests, we will focus on canonical forms that can be compared efficiently in this thesis.

To summarise, a canonical representation of a *CPQ* core query graph is basically a representation of the isomorphic equivalence class of that core that is equal for all cores that are isomorphically equivalent. Essentially, the canonical form is a proxy that makes it possible to check two graphs for equivalence without having to directly compare them. In other words, two graphs are equivalent if their canonical form is equal. It is worth noting that this comes at a cost, canonisation is typically slower than directly testing isomorphic equivalence between two graphs. However, it is important to consider that this cost comes in the form of preprocessing time. After index construction, testing for equality will be extremely efficient.

3.4.1 Nauty

Since canonisation is a key component in the design of index keys, it is important that we make use of the best approach currently available. As noted in Section 3.4 canonisation is closely related to isomorphism testing and is therefore a hard problem to solve efficiently. Fortunately, there is also a wealth of literature available on the topic due to its applications in many fields. In order to find the approach best suited to our use case a prior survey has been conducted of existing canonisation literature [35]. The main focus of this survey was on the performance of algorithms with an existing implementation on *CPQ* query graphs. In this survey, the sparse version of nauty was found to be the algorithm best suited for this purpose. However, as will be further discussed in Section 6.1.2, better options may exist. Next follows a brief history of nauty as also presented in the survey report [35].

Nauty is one of the oldest and most well known algorithms for practical graph isomorphism testing and canonical labelling. The algorithm was first theorised in a master's thesis by Brendan McKay in 1976 [48] with the goal to make labelling graphs with large automorphism groups more efficient. The algorithm was then further developed and eventually published [49, 51]. The core theory behind nauty was based on the refinement-individualization

paradigm proposed by Parris and Read [58], which was later refined by Corneil and Gotlieb [18] and Arlazarov et al. [70]. This approach refines partitions of the vertex set while keeping some vertices fixed. The main difference between the refinement-individualization paradigm and McKay's algorithm was that McKay's algorithm made clever use of automorphisms to reduce the size of the search space. Sometime after the publication of the first paper [51] the proposed algorithm was actually implemented and later became known under the name 'nauty' (**no** **aut**omorphisms, **y**es?).

Currently nauty is the most established and most extensive software available for computing automorphism groups. This is especially true after nauty and a different canonisation algorithm called Traces [60] were merged and became a single software package [54], since Traces covers for some of the shortcomings of nauty making the software package as a whole more versatile. However, instead of only focussing on computing these automorphism groups, the nauty & Traces combined software package also contains many utilities for working with graphs and generating graphs. Nauty is also the software that is used internally by popular larger packages such as Magma [16], SageMath [62] and GAP [27].

Nauty is implemented in two different ways. One implementation is the original version, which is aimed at very dense graphs and uses an adjacency matrix to represent the graph data. This version of nauty is referred to as the dense version of nauty. Later a program called saucy [19] was published, which is a program exclusively focussed on isomorphism detection. Saucy was heavily inspired by nauty, but made use of sparse data structures to represent the graph data. This modification made saucy much more suitable for many real world use cases. In response to this a version of nauty was released that used sparse data structures. This version of nauty is referred to as the sparse version of nauty and is as mentioned the version most suited to handle *CPQ* query graphs.

Both implementations of nauty accept a directed coloured graph as input that is not allowed to contain parallel edges. Recall that *CPQ* query graphs are edge labelled graphs that are allowed to contain parallel edges. For this project that means we need to transform a *CPQ* query graph to a suitable input first before we can use nauty. This transformation process will be covered in Section 3.4.2.1.

Finally, we end with some useful information about nauty and the way it will be used in the implementation of this project. The version of nauty that was evaluated for the survey report was version 2.7. Since a newer version was released on the 16th of November 2022, version 2.8.6 will be used instead to create the index [52]. It is also worth mentioning that nauty & Traces is released under the Apache License v2.0 [3]. Given that nauty is implemented in C while the rest of the index codebase is implemented in Java, a native binding using the Java Native Interface (JNI) [56] will be used to interface with nauty. Lastly, an extensive manual with instructions on how to use nauty is also available [53].

3.4.2 Computing a Canonical Form

The focus of this section will be on the complete process to compute a canonical representation given a *CPQ* query graph as input. How to encode this canonical representation into a form that can be stored and used for equality checks will be covered in Section 3.4.3. First, it is important to note that so far we have been discussing canonisation for testing isomorphic equivalence. However, similar to how we found in Section 3.1 that a regular homomorphism was insufficient, here a regular isomorphism is also insufficient. The root cause again being the lack of information on the special source and target vertices. Fortunately, we can resolve this issue in much the same way as before by introducing the notions of a query isomorphism and query canonisation.

Similar to Section 3.1.1, we will again start from the definition of regular isomorphism as presented in Section 2.2.6. This definition defines isomorphism as an homomorphism that is also bijective. Consequently, we will define a query isomorphism to be a query homomorphism that is also bijective. From this it then follows that query canonisation is simply canonisation based on query isomorphism instead of isomorphism. However, nauty does not make use of query isomorphism and it would be beneficial to use nauty in its unaltered form to simplify updates. Luckily, it is possible to use nauty without modifications by manually tracking the information about the source and target vertex separate from the regular isomorphism information computed by nauty. The details for this metadata tracking process will be covered in Section 3.4.2.4.

In order to clearly showcase the entire canonisation process we will make use of a running example. The *CPQ* of which we will be computing the canonical representation will be $q = (a \circ b) \cap (a \circ c)$ and its query graph is shown in Figure 3.15. For later use each vertex is assigned an arbitrary numerical identifier shown in red, note that these should not be seen as vertex labels.

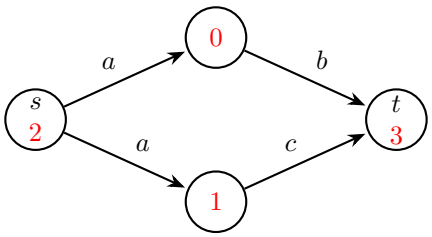


Figure 3.15: Running example query graph for $q = (a \circ b) \cap (a \circ c)$.

3.4.2.1 Graph Transform

As mentioned in Section 3.4.1, nauty cannot directly accept *CPQ* query graphs as input, so we need to transform the input in such a way that any isomorphic equivalence relation is preserved. However, conveniently this problem was already solved as part of the *CPQ* Keys project [35]. To recap, the simplest way to turn a graph with edge labels into a graph without edge labels is to move the edge labels from the edges to newly created vertices. This method has also been suggested on the mailing list for nauty by the original creator of nauty [50].

Formally, given an edge labelled graph $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1, \mathcal{L}_1)$, where the edge set is defined as $\mathcal{E}_1 \subseteq \mathcal{V}_1 \times \mathcal{V}_1 \times \mathcal{L}_1$. We can create a transformed graph $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$, where the edge set is defined as $\mathcal{E}_1 \subseteq \mathcal{V}_1 \times \mathcal{V}_1$ as follows. The vertex set is copied as is $\mathcal{V}_2 = \mathcal{V}_1$, though note that we will be allowing vertex labels in the transformed graph. Then for each edge $(v, u, l) \in \mathcal{E}_1$ we do the following. We first add a new vertex m with label l to $\mathcal{V}_2$ and then add the edges (v, m) and (m, u) to $\mathcal{E}_2$. Since this transform introduces a new vertex for each edge in the original graph and also splits each original edge into two edges, we can say the following about the size of the resulting graph. If an input graph has $|\mathcal{V}|$ vertices and $|\mathcal{E}|$ edges, then the transformed graph has $|\mathcal{V}| + |\mathcal{E}|$ vertices and $2 \cdot |\mathcal{E}|$ edges. An example for this transform is shown in Figure 3.16. Note that as a side effect of this transformation we also solve the issue of parallel edges, as every edge in a *CPQ* query graph has an edge label.

(a) Original labelled input graph.

(b) Transformed output graph.

Figure 3.16: Example for moving edge labels to vertices.

Having introduced the transformation, the next step is to apply it to our example graph from Figure 3.15. The result of applying this transformation can be seen in Figure 3.17. The transformed edge labels are shown in blue.

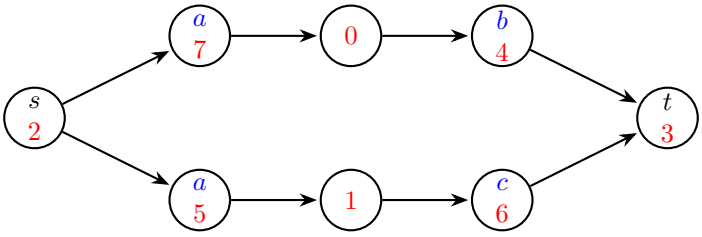


Figure 3.17: Running example query graph after transforming labels to vertices.

3.4.2.2 Coloured Graph Construction

The final step before computing the canonical relabelling of a graph with nauty is completely making the input conform to the expected input format. For nauty that means we need a vertex coloured graph, which is different from the partially vertex labelled graph we have currently. Thus we will assign every regular node without a label a default colour and every other node a colour according to their label. Note that the source and target vertices act like labelled vertices and thus need to be considered as labelled here. If the source and target are not labelled then it is possible for them to be swapped on graphs where all paths between the source and target are fully symmetric. It is also worth noting that internally each colour is simply represented by a numerical identifier. After applying the colouring we end up with the graph shown in Figure 3.18, where the

top number in each node represents the colour and the bottom number the vertex identifier. The legend on the right summarises the colours with their numerical identifier and the label they represent if any.

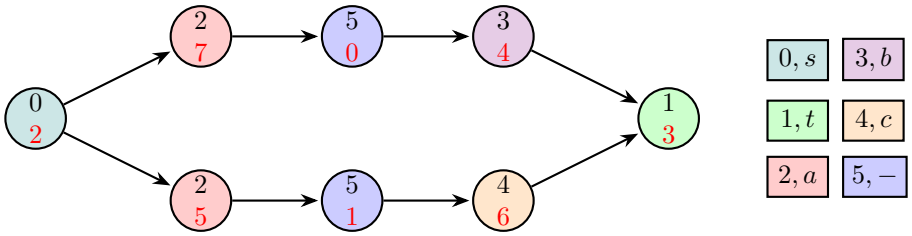


Figure 3.18: Coloured nauty input graph for the running example.

Finally, it is important to note that while colours are distinct, they cannot be distinguished. For example, replacing the label for the colour representing the label c with some completely new label like d results in the exact same input for nauty. Thus, extra metadata to track which colour represents which label is required and will be discussed further in Section 3.4.2.4. However, there is one more important detail to account for. Since nauty cannot distinguish colours and only knows when two vertices have distinct colours, it is extremely important to keep the order in which labels are mapped to colours the same. Internally nauty works by first relabelling the vertices in the colour class with the lowest numerical identifier. This means that the colour classes with a low colour class identifier will always receive the lowest vertex identifiers in the relabelled graph. Therefore, the mapping from label to colour class has to be consistent between calls to nauty. More formally, for labels $a, b \in \mathcal{L}$ with colour identifiers $c_a, c_b \in \mathbb{N}$ and given some strict total order $\prec$ on $\mathcal{L}$, if $a \prec b$ then always $c_a < c_b$. If this property is not preserved then canonical forms may be equal when they should not be, or be different when the original graphs were the same. However, note that this property is fairly trivial to preserve if internally labels are already being represented by some numerical identifier. For the colour classes not directly associated with edge labels we can use any order we want as long as it is consistent. Within this thesis project we will follow the following order:

1) The first colour class always represents the source node.

2) If the target node is different from the source node the second colour class always represents the target node. If the target node is equal to the source node then there is no colour class specifically for the target node and the first colour class represents both the source and target node.

3) Any nodes representing a former edge label $l \in \mathcal{L}$.

4) If there are any vertices without a vertex label they are assigned to the last colour class.

Note that this means that to compare graphs to the example in Figure 3.18, we would always need to use the class order $s \prec t \prec (a \prec b \prec c) \prec -$.

3.4.2.3 Canonical Relabelling

Now that the complete nauty input has been constructed we can use nauty to compute a canonical relabelling of the graph. This canonical relabelling is a mapping from the current vertex identifiers to a new vertex identifier for each vertex. Applying this relabelling gives every vertex in the graph a canonical vertex identifier resulting in a canonically labelled graph. For example, for the graph in Figure 3.18 nauty gives the output in Listing 3.1.

Listing 3.1: Graph relabelling output from nauty.

```
[2, 3, 5, 7, 4, 6, 1, 0]
```

This output array should be seen as a mapping function that should be read as follows. The vertex at position i in the array formerly had the identifier at that position in the array. For example, for the first index the output states that the vertex that should now be labelled with 0 was formerly labelled with 2. This leads to the remapping shown in Equation 3.13, where the left hand side denotes the old identifier of a vertex and the right hand side the new identifier.

$$\begin{aligned} 0 &\to 7 \\ 1 &\to 6 \\ 2 &\to 0 \\ 3 &\to 1 \\ 4 &\to 4 \\ 5 &\to 2 \\ 6 &\to 5 \\ 7 &\to 3 \end{aligned} \tag{3.13}$$

Applying this mapping to the running example graph we get the graph shown in Figure 3.19.

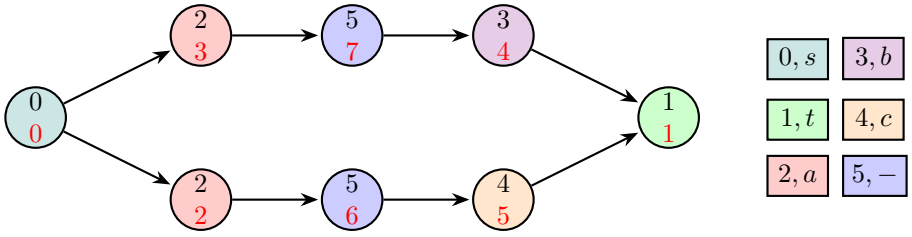


Figure 3.19: Running example after being canonically relabelled.

3.4.2.4 Required Metadata

Having computed the canonically labelled graph with nauty we now have all the information required to construct a complete canonical form. This section will be used to give a complete overview of all the information we need to encode, methods to encode the information will be discussed in Section 3.4.3.

1) **Graph**: The structure of the created canonically labelled graph has to be encoded. This comes down to encoding the adjacencies of all vertices in some way, for example using an adjacency list representation of the graph.
2) **Labels**: We need to encode which vertices represent certain labels from $\mathcal{L}$.
3) **Source**: We need to encode which vertex is the source vertex.
4) **Target**: We need to encode which vertex is the target vertex.
5) **Unlabelled**: We need to encode which vertices have no label.

Note that some information could be encoded implicitly. For example, unlabelled vertices can uniquely be determined if the complete set of vertices and the set of labelled vertices are known.

3.4.3 Representation of a Canonical Form

Now that all the components required for a canonical form have been laid out in Section 3.4.2.4, the next step is to design a convenient method to encode all the information. It is important to note that there is no real correct or wrong answer to this question. However, we can strive for specific goals when designing an encoding. Since the index will need to store a lot of cores, the main goal we will optimise for is compactness. Nevertheless, we will first start by presenting a convenient human readable encoding in Section 3.4.3.1 that is more suitable for debugging. Note that the main thing to pay attention to when designing an encoding is the reversibility of the encoding. When the original information can be recovered unambiguously from the encoded representation, the encoding is suitable for our use case. However, note that it is fine if decoding requires us to supplement information that never changes.

Before discussing the designed formats in Section 3.4.3.1 and 3.4.3.2, we will first discuss two optimisations that are used by both formats. Both optimisations make use of explicitly specified information to implicitly define other information. Next both optimisations will be discussed in more detail:

1) **Unlabelled Vertices**: The first optimisation is rather straight forward. Since we already need to include information in the canonical representation about which vertices have which labels, there is no need to also include which vertices have no label. This optimisation does rely on the fact that vertex identifiers are sequential and start from 0 and that we know how many vertices are in the graph. Note that we know the number of vertices because we need to encode graph adjacencies.

2) Labelled Vertices: After canonically relabelling a graph, note that all vertices with a specific label have sequential identifiers. This means that instead of listing each vertex we only need to store the start and end of the range. Moreover, since all label sequence ranges are sequential we can get away with only storing how many vertices have a given label. Note that we can only do this because there are also no gaps in the vertex identifiers. Furthermore, note that this optimisation does not conflict with the first optimisation since we can still uniquely reconstruct the identifiers of vertices with a specific label.

3.4.3.1 Human Readable

In this section we will introduce a relatively simple encoding of a canonical form that is primarily useful for debugging due to being human readable. The core idea is to essentially just list everything directly with the only optimisations being the ones mentioned in Section 3.4.3. Firstly, the source and target vertex are listed. Secondly, all the numerical identifiers for each label prefixed by 'l' are listed together with the total number of vertices with that label in the graph. Finally, the adjacencies of the graph are listed where the identifier of the source vertex is prefixed by 'e' and all the target vertices are listed as the value. An example of this encoding for the running example graph from Figure 3.15 in Section 3.4.2 is given in Listing 3.2.

Listing 3.2: Human readable canonical form example.

```
s=0,t=1,l0=2,l1=1,l2=1,e0={2, 3},e1={},e2={6},e3={7},e4={1},e5={1},e6={5},e7={4}
src trg labels                  edges
```

Note that there are a number of places where order is extremely important in this encoding. First of all, labels should always be listed in the order that is consistent with the order that was chosen in Section 3.4.2.2. Secondly, the adjacencies for each vertex need to be sorted, this is important to make sure that we can compare canonical forms by checking equality.

3.4.3.2 Binary Format

Having introduced a readable format in Section 3.4.3.1, the goal of this section is to work towards a representation that is as compact as possible. As a metric of compactness we will use either bytes or bits and for concrete comparisons we will use the same example graph from Section 3.4.3.1. To set a baseline we will first compute how many bytes were used by the human readable representation. Note that the canonical string for the example in Listing 3.2 is 79 characters long, assuming the use of UTF-8 this equates to 79 bytes or 632 bits. Moreover, note that storing numbers in string form is extremely inefficient. However, also note that the form values readability, hence some possible optimisations were not performed.

The goal of this section is to propose an alternative binary format, such that we can store numbers more efficiently. At its core this format will still encode the same information as the human readable format. The most notable addition is the addition of a field at the start indicating the total number of vertices. This field may seem redundant, and in fact it currently is. However, this field will be important for some of the optimisations in the latter part of this section. The format of the encoding is shown in Table 3.7.

Table 3.7: Binary canonical representation format.

Canonical Form											
Vertices	Source	Target	Labels				Edges				
			n	Label		$\cdots$	Vertex			$\cdots$	
				l_{id}	n		n	v_{id}	$\cdots$		

In this table the dots ($\cdots$) indicate that the previous cell can be repeated 0 or more times. We will explain all the fields in detail next:

1) Vertices: This is the total number of vertices in the graph. Given the nature of how vertex identifiers are assigned, this number tells us exactly what the identifiers are for all vertices in the graph.

2) Source: This is the vertex identifier of the source vertex.

3) Target: This is the vertex identifier of the target vertex.

4) Labels: This segment stores all information related to labels.

- n: This number indicates how many labels there are in the graph and thus also how many labels are about to follow in the encoding.

- Label(l_{id}, n): Each label is stored as two numbers, the first representing the numerical identifier of the label l_{id} and the second indicating the number of vertices in the graph with the label n.

5) Edges: This segment stores the graph by storing all information related to vertex adjacencies, information is present for every vertex in order of vertex identifier. Note that it is important to sort the vertices.

- Vertex($n, v_{id}, \dots$): For each source vertex the identifiers v_{id} of the vertices this vertex has an edge to are stored. The total number of out going edges is indicated by n.

Following this specification we can represent the information for the example graph from Figure 3.15 in Section 3.4.2 as shown in Listing 3.3. Note that there is a difference in how labels and adjacencies are encoded. For labels we choose to omit any labels not present in the graph, while we list all vertices, even those without out going edges. This distinction was made because a query graph may not use all labels, but because it is a *CPQ* most of its vertices will have edges.

Listing 3.3: Numerical canonical form example.

```
8 0 1 3 0 2 1 1 2 1 2 2 3 0 1 6 1 7 1 1 1 1 1 5 1 4
v s t n l n l n l n n e   n n e n e n e n e n e n e
          10   11   12   v0   v1 v2   v3   v4   v5   v6   v7
          labels         edges
```

Computing the storage requirement for this canonical from is rather straightforward, the representation consists of 26 integers. Assuming the most common method of storage using 32 bit integers, this leads to 104 bytes of storage or 832 bits. Note that this is worse than the human readable canonical form. However, note that 32 bits is way more than we strictly require, hence significant reductions in storage space are still possible. To achieve these reductions we first need to determine realistic bounds on the sizes of all the numbers we want to store. After doing this we can propose a proper variable length encoding for the canonical form. For this, note that all numbers fall in one of two categories:

1) Firstly, we have numbers that are somehow related to the size of the vertex set $\mathcal{V}$ of the canonical graph. This category includes the vertex set size, any number representing the identifier of a vertex, and any number representing a quantity of vertices. Since all vertex identifiers are less than $|\mathcal{V}|$ we know that $\log_2(|\mathcal{V}|)$ bits will always suffice to store a vertex identifier or vertex quantity.

2) Secondly, we have numbers that are somehow related to the size of the label set $\mathcal{L}$ of the database graph. This category includes the label count and the numbers representing label identifiers. Similar to the vertex set case, we know that $\log_2(|\mathcal{L}|)$ bits will always suffice to store label information.

Using this information we can divide the fields from the canonical representation in Table 3.7 into two categories:

- **Vertex Bound**: Vertices, Source, Target, Labels.Label.n, Edges.Vertex.n, Edges.Vertex.v_{id}.
- **Label Bound**: Labels.n, Labels.Label.l_{id}.

The next step then comes down to deciding how the size information for these two categories will be provided. For this thesis project it was decided to use a scheme where a configurable but otherwise fixed number of bits is allocated to each of these categories. This approach makes the most sense for the label set, but does require that the label set of the graph is not extended at runtime. Optionally some extra space could be allocated to accommodate a limited amount of growth.

Bounding the vertices in this way is slightly more complicated. While possible, it does not make sense to use the total number of vertices in the graph as a bound. The majority of input queries, especially when restricted to a specific *CPQ* diameter, will not have a query graph vertex count that is of a similar magnitude to the number of vertices in the entire database graph. Moreover, it is likely that the number of vertices in the database graph changes at runtime. Hence the solution that will be used in this thesis project is to encode the number of bits to use for vertices in the 'Vertices' field of the canonical form. We will arbitrarily allocate a fixed number of bits to encode the vertices field and then using this field compute how many bits are used to encode other vertex bound fields in the canonical representation. Even if we decide to allocate a lot of bits for the 'Vertices' field this method would still only require us to pay that price once for each computed canonical form. Furthermore, it is likely that a good bound can be found after analysing executed database queries for a period of time.

Using this variable length encoding scheme we can revisit the canonical form example in Listing 3.3. For this example we will allocate 5 bits to labels, allowing up to $2^5 = 32$ graph labels and 10 bits to vertices, allowing up to $2^{10} = 1024$ vertices in a single canonically labelled graph. For the example these bounds are clearly over

provisioned. Going over the example canonical form, we find 21 vertex bound fields and 4 label bound fields. Since the encoded vertex count is 8, this means the variable width for vertex bound fields is $\lceil \log_2(8) \rceil = 3$ bits. In total this means we need $10 + 4 \cdot 5 + 21 \cdot 3 = 93$ bits or 11.625 bytes. In practise we will simply pad with 0 bits until the nearest full byte giving 12 bytes in total. However, it is clear that this variable width encoding is far more efficient that the other two methods we have discussed and thus this is also the method that will be used in the *CPQ*-native Index. The complete binary canonical form for the running example from Figure 3.15 in Section 3.4.2 is shown in Listing 3.4.

Listing 3.4: Binary canonical form example.

```
00000010 00000001 00011000 00010000 01001000 10001010
01001100 00011100 01111001 00100100 10011010 01100000
```

Note that this format is perhaps not the most convenient to use in writing or debugging. Hence, under certain circumstances it can be beneficial to encode the bits as Base64 [42]. Doing so would give the canonical form as `AgEYEEiKTBx5JJpg`.

3.5 Summary

In this chapter we have presented the foundations for the *CPQ* cores that will be used as keys in the *CPQ*-native Index. The two main topics that were discussed in this chapter were the computation of *CPQ* cores and computing canonical representations of *CPQ* cores that are suitable for use in a graph database index. In pursuit of these goals we have introduced the notion of a query homomorphism, developed a new tree decomposition algorithm, modified an existing homomorphism test to support query homomorphism, formalised the definition of a *CPQ* core, shown an intuitive method to compute *CPQ* cores, developed a novel algorithm for core computation, and developed a method for computing and representing canonical forms of cores. Together all these components present an answer to the first sub-question posed in Section 1.2 of how we can efficiently compute a CPQ_k core. Building on these concepts we will next introduce the *CPQ*-native Index in Chapter 4.

The *CPQ*-native Index

4 第四章

In this chapter we will introduce the complete design for the *CPQ*-native Index. The goal of the index is to partition all paths in the graph by the *CPQ* cores that match them. Using this approach we can then immediately look up for any given *CPQ* core which paths it matches, without having to evaluate the *CPQ* core as a query on the graph. The topics that will be discussed in this chapter make heavy use of the *CPQ* core theory introduced in Chapter 3 and the language-aware indexing paper [65]. Note in particular that this paper already solved the partitioning problem by using k-path-bisimulation. Also recall that we formally introduced the definition of k-path-bisimulation in Section 2.2.5. Essentially our challenge with the *CPQ*-native Index is augmenting the approach presented in the paper with the required logic to compute *CPQ* cores for each partition block.

During this chapter we will make use of a simple running example graph given in Figure 4.1. This graph represents all the vertices and connections that are present in the database and we will show how to compute a *CPQ*-native Index for it. In order to compute the index, we will first show in Section 4.1 how to use the definition of k-path-bisimulation from Section 2.2.5 to compute the k-path-bisimulation partitioning of the running example graph. Using the computed partitioning we will then show how to compute the *CPQ* cores associated with each partition block of the index in Section 4.2. Note that incidentally the database graph in Figure 4.1 is also structured like a valid *CPQ*, for example, the query graph for $a \cap (a \circ b)$ has the same structure. However, it is not a requirement for index construction that the database graph is itself structured like a valid *CPQ*.

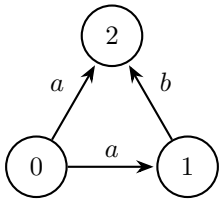


Figure 4.1: Running example database graph for index construction.

4.1 Graph Partitioning

Partitioning a graph using k-path-bisimulation is an incremental process where blocks are continually refined into smaller blocks for a larger value of k. Note that this refinement procedure is based on paths of length k and follows the definition of k-path-bisimulation, recall from Section 2.2.5 that this means that no CPQ_k can distinguish the paths within a block. The refinement of these blocks can then happen following the third rule for k-path-bisimilarity, which extends paths by another path to reach length k. Note that due to the structure of paths, it does not matter how we build a longer path. For example, extending ab with c results in the exact same path as extending a with bc, both giving abc. This means that the definition for k-path-bisimulation is more permissive than we need in this case. The language-aware index [65] takes advantage of this fact and always extends a path of length $k-1$ with a path of length 1 to reach length k. More concretely, this means that we start by computing the 1-path-bisimulation partitioning of the graph and we then further refine this partitioning for the 2-path-bisimulation partitioning. Similarly, the 3-path-bisimulation partitioning is computed from the 2-path-bisimulation partitioning by extending paths by one additional label. Larger values of k are computed in a similar fashion.

While the strategy outlined above works perfectly for a path based index, it falls short for a *CPQ*-native Index. Note that the core assumption for this strategy relies on the fact that we can always construct blocks for length k from a block for length $k-1$ and a block for length 1. However, for the *CPQ*-native Index we do not only want to compute paths, we also want to compute cores, and unfortunately this assumption does not hold for *CPQ*s. For example, if we want to construct the diameter four *CPQ* $((a \circ a) \cap (b \circ b)) \circ ((c \circ c) \cap (d \circ d))$, we cannot do this by joining a diameter three and a diameter one *CPQ*. The only way to construct this *CPQ* using a join

is by combining a diameter two *CPQ* with another diameter two *CPQ*. This suggests that for the index we need to consider any combination of blocks from previous layers, unlike the path based reference work. However, note that this does not affect the created partitioning, we are not changing the definition of k-path-bisimulation based partitioning. We are only computing some extra information that was not required for the path based language-aware index. However, note that we are increasing the runtime of the partitioning procedure. Recall from Section 2.1.5 that the partitioning procedure originally took polynomial time for the language-aware index [65]. As we will show in Section 4.1.2, we will be adding a factor k to the runtime complexity.

4.1.1 Partitioning for 1-path-bisimulation

Partitioning a graph based on 1-path-bisimulation means partitioning the graph based on all paths of length 1. From the definition of k-path-bisimulation introduced in Section 2.2.5, this means that only the first two conditions apply. More specifically, this means that essentially all paths are partitioned by their label and whether or not they are a self loop. Note that including reverse edges, there are only six pairs in the example graph from Figure 4.1 corresponding to all length one paths, namely $(0,1),(0,2),(1,0),(1,2),(2,0)$ and $(2,1)$. Also note that there may be pairs in the graph that are not connected at length one. If present, these paths will simply not appear in the partitioning until a sufficiently large value of k is used. Technically these paths exist together in an index block representing paths that are not connected at length one. Later refinements for larger values of k can split off new blocks from this block when pairs do become connected. Note that we do not explicitly store a block containing unconnected pairs as this would unnecessarily use up a lot of storage. Instead we find newly connected paths by combining existing blocks as we will discuss in Section 4.1.2.

To construct the 1-path-bisimulation partitioning, the first step is to list all the paths in the graph that exist at length one. This information can easily be derived from edges in the graph, with each edge contributing two paths. Note that some paths may be found more than once using this method, care should be taken to remove duplicate paths. The next step is then to label each found path with the labels that appear between the source and target node on all edges that connect these vertices. The result of doing this for the running example graph from Figure 4.1 can be found in Table 4.1. Note that for the example graph each pair only has one label, in practice there could be more than one label for a given path.

Table 4.1: All paths and their labels for $k = 1$.

Path	**Labels**
$(0,1)$	a
$(0,2)$	a
$(1,0)$	a^-
$(2,0)$	a^-
$(1,2)$	b
$(2,1)$	b^-

After compiling all the paths and their labels the next step is to sort by label and loop status, such that paths with identical labels and cyclic properties are listed next to each other. More explicitly, when sorting we first compare the labels, then the loop status, then the source vertex of the path, and finally the target vertex of the path. Any total ordering for the labels would work, in this thesis we will use a lexicographical order where inverted labels are placed directly after their regular counterpart. Applying this ordering to the data in Table 4.1 gives the sorted data in Table 4.2. Consecutive paths with the same labels and loop status are then assigned to the same block. Note that there are no self loops in the example graph, so only the labels matter.

Table 4.2: All paths and their labels sorted and with block identifier for $k = 1$.

Block	**Paths**	**Labels**
1	$(0,1)$	a
	$(0,2)$	a
2	$(1,0)$	a^-
	$(2,0)$	a^-
3	$(1,2)$	b
4	$(2,1)$	b^-

Using this data we can visualise the index for $k = 1$ as shown in Figure 4.2.

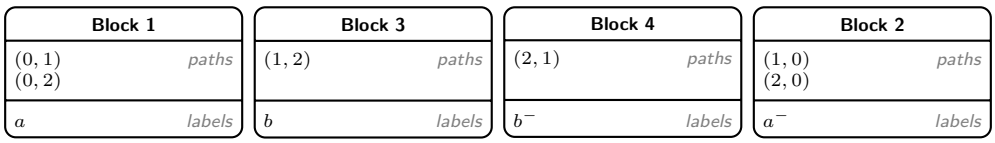


Figure 4.2: The first layer of the index partitioning.

4.1.2 Partitioning for k-path-bisimulation

Now that we have built a set of initial blocks for length one, the next step is to use these to incrementally build blocks for larger values of k. Note that following the last condition in the definition for k-path-bisimulation in Section 2.2.5, we need to combine all possible combinations of blocks for a shorter length to form a block for length k. As mentioned in Section 4.1, the path based index could avoid computing all combinations. However, for a *CPQ*-native Index we need all this information later for core computation, which will be explained in Section 4.2. Note that the number of combinations that sum to a given value increases linearly, for k up to 9 all the required combinations are shown in Equation 4.1. In general, the combinations can be quantified using $\{(i,j) \mid k = i + j \wedge i,j \in \mathbb{N}^+\}$, observe that this can easily be implemented in practice using a simple loop with $i \in \{1,\ldots,k-1\}$ and $j = k - i$. Also note that this means that we are adding an extra factor k to the runtime complexity, which was originally already polynomial as mentioned in Section 2.1.5.

$$
\begin{aligned}
&k = 1 : -\\
&k = 2 : 1 \circ 1\\
&k = 3 : 1 \circ 2, 2 \circ 1\\
&k = 4 : 1 \circ 3, 2 \circ 2, 3 \circ 1\\
&k = 5 : 1 \circ 4, 2 \circ 3, 3 \circ 2, 4 \circ 1\\
&k = 6 : 1 \circ 5, 2 \circ 4, 3 \circ 3, 4 \circ 2, 5 \circ 1\\
&k = 7 : 1 \circ 6, 2 \circ 5, 3 \circ 4, 4 \circ 3, 5 \circ 2, 6 \circ 1\\
&k = 8 : 1 \circ 7, 2 \circ 6, 3 \circ 5, 4 \circ 4, 5 \circ 3, 6 \circ 2, 7 \circ 1\\
&k = 9 : 1 \circ 8, 2 \circ 7, 3 \circ 6, 4 \circ 5, 5 \circ 4, 6 \circ 3, 7 \circ 2, 8 \circ 1
\end{aligned}
\tag{4.1}
$$

It is important to note that '$\circ$' is not commutative in Equation 4.1, this stems from the fact that we will be using the *CPQ* join operation '$\circ$' to combine *CPQ*s for blocks later, which is not a commutative operation. For a more intuitive understanding of the given equations, the equations in Equation 4.1 should be interpreted as follows:

- For $k = 1$, there is no need to combine blocks, this layer is computed as described in Section 4.1.1.
- For $k = 2$, the only way to make a block for length 2 is to combine two blocks for length 1.
- Starting with $k = 3$, we need more than one combination of blocks from previous layers to form the next layer. Also note here that $1 \circ 2$ and $2 \circ 1$ do not encode the same information. For example, the diameter three *CPQ* $a \circ ((b \circ b) \cap (c \circ c))$, can only be constructed using a join by joining the diameter one *CPQ* a with the diameter two *CPQ* $(b \circ b) \cap (c \circ c)$.

Having explained the fundamentals, we will next explain how to compute a new layer of blocks. It is worth noting that while the general process resembles the approach to computing 1-path-bisimulation, there are some important differences. Similar to before we will again list all the paths in the graph that exist at length two, namely $(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1)$ and $(2,2)$. Note that all possible pairs are reachable at length two in the example graph, so the block containing unreachable pairs no longer exists and will instead be fully refined into blocks containing reachable pairs. Next we will also attach an ancestor to each path, this ancestor represents the previous layer block a path was in. Pairs that were not connected before do not have an ancestor. Strictly speaking their ancestor would be the block containing unconnected pairs, but we do not consider this block as no label sequences match the paths in this block, meaning it has no relevant information for query evaluation. Finally, we will list for each path which blocks the paths came from that were combined to form it. For example, in the example graph the path $(0,1)$ was constructed by extending the path $(0,2)$ from block 1 with the path $(2,1)$ from block 4. Note that the ancestor is essentially a compact representation of these combinations summarising previous layers without needing to repeat them explicitly. For the example graph Table 4.3 tabulates all the paths with their ancestor, combinations and labels.

Table 4.3: All paths with their ancestor and combinations for $k = 2$.

Path	**Ancestor**	**Combinations**	**Labels**
$(0, 0)$	-	$1 \circ 2$	aa^-
$(0, 1)$	1	$1 \circ 4$	a, ab^-
$(0, 2)$	1	$1 \circ 3$	a, ab
$(1, 0)$	2	$3 \circ 2$	a^-, ba^-
$(1, 1)$	-	$2 \circ 1, 3 \circ 4$	a^-a, bb^-
$(1, 2)$	3	$2 \circ 1$	b, a^-a
$(2, 0)$	2	$4 \circ 2$	a^-, b^-a^-
$(2, 1)$	4	$2 \circ 1$	b^-, a^-a
$(2, 2)$	-	$2 \circ 1, 4 \circ 3$	a^-a, b^-b

Note that the label sequences can be derived directly from the combined blocks and the ancestor block. However, the label sequences do not need to be computed at all for the *CPQ*-native Index and are only listed to highlight how the path-based index computes them. After attaching all this extra information to the paths the next step is again to sort all the paths, such that k-path-bisimilar paths are listed sequentially. When sorting the following properties should be compared:

1) Ancestor

2) Combinations

3) Loop Status

4) Path Source Vertex

5) Path Target Vertex

The exact order used to compare these properties does not matter, we only need paths with identical properties to be listed sequentially. However, the listed order is used by the reference implementation for this thesis and some comparisons are more expensive than others. For example, since combinations are most likely to differ, but also the most expensive to compare, we use a hash value computed from the combinations for comparison first before comparing the actual combinations. If such an optimisation is not employed, then comparing combinations last is a more performant option. However, note that comparing a property that is likely to differ first is beneficial for sorting performance.

After sorting we then obtain the order shown in Table 4.4. Sequential paths with the same combinations and loop status are assigned the same block identifier. Note that for the example graph none of the paths are equivalent under 2-path-bisimulation, so each block only consists of a single path.

Table 4.4: All paths sorted and with block identifier for $k = 2$.

Block	**Path**	**Ancestor**	**Combinations**	**Labels**
5	$(0, 0)$	-	$1 \circ 2$	aa^-
6	$(0, 2)$	1	$1 \circ 3$	a, ab
7	$(0, 1)$	1	$1 \circ 4$	a, ab^-
8	$(1, 2)$	3	$2 \circ 1$	b, a^-a
9	$(2, 1)$	4	$2 \circ 1$	b^-, a^-a
10	$(1, 0)$	2	$3 \circ 2$	a^-, ba^-
11	$(2, 0)$	2	$4 \circ 2$	a^-, b^-a^-
12	$(1, 1)$	-	$2 \circ 1, 3 \circ 4$	a^-a, bb^-
13	$(2, 2)$	-	$2 \circ 1, 4 \circ 3$	a^-a, b^-b

Using this data we can visualise the index for $k = 2$ as shown in Figure 4.3. Note that the blocks for $k = 1$ are only shown for extra clarity, they are not actually required in the constructed index. The arrows are used to indicate ancestor relationships between blocks. These arrows point from an ancestor block to all the blocks that were refined from it. Note that it is possible for blocks to be refined without splitting into multiple blocks.

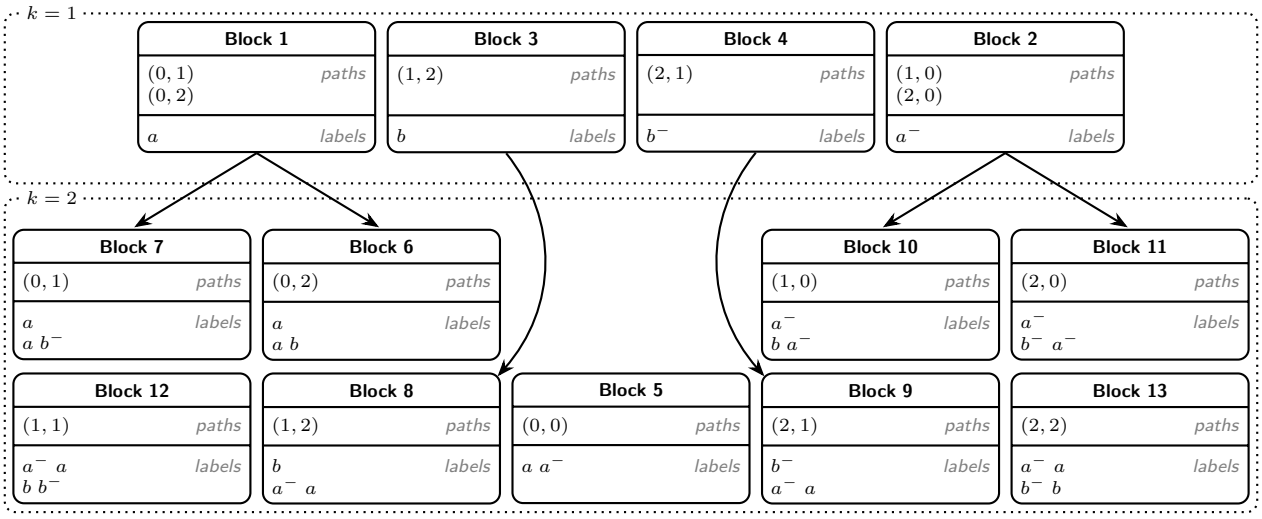


Figure 4.3: The fully constructed index for $k = 2$.

From this figure it is clear that the construction of the index blocks happens in three ways:

1) Some blocks are just copied from a previous layer, possibly with added longer label sequences.

2) Some blocks appear to be completely new, these represent paths that were not connected before under a lower value of k. Thus technically these blocks represent refinements of the block containing all unconnected paths and the pairs in them first became connected for the current value of k.

3) Some blocks from the previous layer are split into multiple blocks in the new layer, with each block receiving some of the paths in the original block. This happens when paths cannot be distinguished by any query in CPQ_{k-1}, but can be distinguished by a query in CPQ_k. Blocks created in this fashion are also why index partitioning is called a refinement procedure.

4.2 Mapping Cores to Blocks

After graph partitioning we now have a partitioning of all paths in the graph of length at most k. Each partition block contains pairs with the source and target of each stored path and in addition we stored the length one label sequences that map to a partition. For blocks that are refinements of a block in a previous layer, we also store a reference to this ancestor block. The next step is to compute cores for all the blocks layer by layer, for which we will closely follow the definition of a *CPQ* and compute candidate cores. All of the candidate cores we will be computing are *CPQ*s that are potentially a core for a block. We will compute a canonical core for these candidate *CPQ*s following the theory discussed in Chapter 3 and save them if they are newly found cores. Computation of these candidates is done in a number of stages, which we will discuss in more detail in Sections 4.2.1, 4.2.2, 4.2.3, 4.2.4 and 4.2.5. After all cores have been computed we will show how to fully complete the index in Section 4.3.

4.2.1 Inherited Cores

Starting with inherited cores, for some blocks there are cores that do not need to be computed at all. These are all the cores that map to the ancestor for a block if the block has an ancestor. Trivially, if these cores matched the paths before, then they still do. Reusing these cores also means we do not need to compute these lower diameter cores again. For a block we will denote the potentially empty set of inherited cores with $\mathcal{C}_{inherit}$.

4.2.2 First Layer Cores

The first layer of the index is the only layer that was not constructed by combining blocks that already existed. In other words, this layer can be viewed as the base case for index construction. However, computing the initial set of cores for this layer is fortunately trivial. Note that all the label sequences for a block are valid *CPQ*s. While we will not make use of this fact for higher values of k, we will compute the initial cores for the first layer from the labels. This initial set effectively corresponds to all paths of length one and we will denote it with $\mathcal{C}_{label}$. However, there may be other cores that map to a block in the first layer of the index. These remaining cores would be generated either via the identity or intersection operation and will be covered in Section 4.2.4 and 4.2.5. Note that the join operation can never appear in a block for $k = 1$ as joining two cores by definition

generally results in a larger diameter. However, there is one exception to this rule in the form of a join with identity. For example, the *CPQ* $a \circ id$ contains a join operation that does not result in an increased diameter. Note though that this *CPQ* is structurally identical to the *CPQ* a. Essentially, a join with identity is an action that does nothing, making it an irrelevant case we do not need to consider. As soon as both sides involved in a join have at least one edge label operation the diameter will increase, meaning such a core will never appear in the first layer.

Incidentally, note that since each block in the first layer of the example graph index contains only one label and none of the blocks contain looping paths, the label conversion cores are the only cores relevant for this layer. This means that the entire example graph index for $k = 1$ can be visualised as shown in Figure 4.4.

Block 1		Block 3		Block 4		Block 2	
$(0,1)$ $(0,2)$	*paths*	$(1,2)$	*paths*	$(2,1)$	*paths*	$(1,0)$ $(2,0)$	*paths*
a	*labels*	b	*labels*	b^-	*labels*	a^-	*labels*
a	*cores*	b	*cores*	b^-	*cores*	a^-	*cores*

Figure 4.4: The first layer of the *CPQ*-native Index with cores.

It is also worth noting that in rare cases first layer blocks do not stay only at the first layer. As mentioned at the end of Section 4.1.2, new blocks are either copied, completely new or split. When a block is copied it often has new label sequences added to it of a longer length, but in some cases no longer label sequences map to a block. These blocks will therefore appear in higher layers of the index without ancestor and without block combinations. During core computation these blocks should therefore be treated as first layer blocks, this also means the join cores that will be covered in Section 4.2.3 do not apply to them.

Finally, we will strengthen our claim that the label sequences make up the initial set of cores by proving that all distinct sequences are a unique core. Note that this fact means that we can completely skip core computation for these core candidates. Also note that this proof is slightly stronger than what we need for the approach we are using, since we only have label sequences of length one.

Theorem 8. *Every distinct label sequence is a unique* CPQ *core (no label sequences share the same core).*

Proof. We will start by proving that the core of any label sequence *CPQ* is in fact that label sequence itself. For this first recall from Section 3.2 that a *CPQ* core is a *CPQ* query graph from which no more edges can be removed without making the graph no longer query homomorphic to itself. Note that by definition there is only a single unique path from source to target in a label sequence. This means that removing any edge will disconnect the graph and break any query homomorphism. Thus, the query graph has to be a *CPQ* core.

For the uniqueness property, note that the *CPQ* for the label sequence is essentially a chain of vertices where each linking edge is uniquely determined by one of the labels in the label sequence. Naturally, this is obviously the case for different labels, but an inverted label will result in a different edge direction. Both of these cases therefore lead to a different unique core. □

4.2.3 Join Cores

Similar to how converted label sequences formed the initial set of cores for the first layer of the index, cores constructed with the join operation essentially form the basis for most blocks with $k > 2$. Recall from Section 4.1.2 that higher layer blocks have a set of combinations of blocks that were used to construct the stored paths. We will make use of this set of combinations $\mathcal{B}_{combinations}$ to compute all the join cores for a block. The general approach is to simply for each combination of blocks join all cores in the first block with all cores in the second block, but note that the block order is important here. The general approach for computing this set of cores is then shown in Equation 4.2.

$$\mathcal{C}_{join} = \{q_1 \circ q_2 \mid q_1 \in A \wedge q_2 \in B \wedge (A, B) \in \mathcal{B}_{combinations}\} \tag{4.2}$$

Note that by definition all the cores created in this way are of at most diameter k. This is due to the fact that the diameters for the blocks in each pair sum to exactly k. However, cores with a diameter less than k can still be generated as not all the cores in a block have the same diameter as the block. Ideally, we would like to prove again that all the core candidates generated in this way are already cores. Unfortunately, while this claim might appear to be true, it does not always hold up.

Theorem 9. *The join of two* CPQ *cores is itself not always a* CPQ *core.*

Proof. We can easily prove this statement by giving a counter example where the join result is not a core. Consider the *CPQ* core $a \cap id$, computing the join of this *CPQ* with itself we get $(a \cap id) \circ (a \cap id)$. Note that this is not a core as the self loop a is repeated twice on the same vertex. The actual core of this graph would be $a \cap id$. □

The main problem with cores generated from joins is that the identity operation presents a problem. Note that we can generally think of the join operation as extending a path with another path. However, the diameter of the identity operation is zero, and here we run into a similar problem to the case discussed in Section 4.2.2. The diameter of a *CPQ* only gives information on the longest path through a *CPQ*. However, shorter paths through a *CPQ* can also exist and the case where the length of the shortest path is zero due to the identity operation leads to a problem for joins where, since not all paths are extended, they may be duplicated instead. Unfortunately, this means that we cannot simply assume join cores to be actual cores, meaning we need to validate them by computing their actual core before before adding them to the index. Note that there is a way to mitigate the effect to some extent. If the first core has no loops on its target vertex and the second core has no loops on its source vertex, we can safely join them to create a new core.

Theorem 10. *The join of two* CPQ *cores is itself a* CPQ *core if the first core has no loops on its target vertex and the second core has no loops on its source vertex.*

Proof. First note that joining any two *CPQ* cores will result in a *CPQ* that has a larger diameter than either of the two original *CPQ*s. The only exception is joining a *CPQ* with identity. However, since joining a *CPQ* with identity changes nothing about the *CPQ* and given that the *CPQ* was originally a core, the result is still a core. Going back to the more common case, increasing the diameter of the *CPQ* means that there are now paths from source to target that do not exist in either of the original two cores. Thus it remains to show that there are no paths that are duplicated and thus would be removable. Intuitively, the first core in the join is essentially a collection prefixes for paths and the second core a collection suffixes. Furthermore, since the two *CPQ*s being joined are cores, these prefixes and suffixes are also all unique. It then remains to prove that none of these prefixes or suffixes can be removed in the newly created *CPQ*. This follows from the fact that any path has to go through the single node that the two *CPQ*s were joined at. Essentially, all paths start at the source, then visit the join node, and finally go to the target node. One thing that could clearly lead to issues here is if there were loops on this join node, which is why we explicitly excluded cases like this.

Assuming that we can remove some part of one of the prefixes, this means that there are now fewer paths leading to the join node. However, this is problematic, if the removed structure is covered by a different prefix path, then we need to conclude that the first core was not actually a core. However, if it was not covered, then we now have fewer paths from source to target, which means that the assumption that we could remove this part was false, leading to a contradiction. A similar argument can be made for the case where we assume we can remove some part of one of the suffixes instead.

Thus we conclude that the join of these two *CPQ*s has to be a core itself. □

It is worth noting that we did not end up implementing this optimisation in the reference index as computing join cores does not take a lot of time compared to intersection cores, as will be discussed in more detail in Section 4.2.4.

4.2.4 Intersection Cores

Intersection cores are the main contributor to the total core count in the *CPQ*-native Index. Note that up to this point we have essentially been computing *CPQ*s mostly consisting of a single main path. Trivially, if we take two of these paths and intersect them, then we obtain a new *CPQ* that also matches the pairs in the same block. This follows directly from the fact that all the *CPQ*s for a block match all the path pairs stored at that block. Unfortunately, this also means that the intersection of any subset of the cores we have computed so far is a potential core candidate for the index block. Assuming we have computed n cores so far, this means we need to compute another 2^n, instantly bringing the runtime of the entire computation to at least exponential.

Next we will cover how to compute the cores with top level intersections for a block. Before covering this it is worth mentioning a special case we have if the block we are computing has an ancestor. Any cores in the set of inherited cores have already been processed by this algorithm, hence it is not required to intersect cores from this set with each other again. This means that when computing intersection cores at most one inherited core is allowed to participate in a new intersection. Following this we can define the set of intersection cores as shown in Equation 4.3.

$$\begin{aligned}\mathcal{C}_{intersect} = &\{q_1 \cap \cdots \cap q_n \mid \{q_1, \ldots, q_n\} \subseteq (\mathcal{C}_{label} \cup \mathcal{C}_{join}) \wedge n \geq 2\} \\ &\cup \{q_0 \cap q_1 \cap \cdots \cap q_n \mid q_0 \in \mathcal{C}_{inherit} \wedge \{q_1, \ldots, q_n\} \subseteq (\mathcal{C}_{label} \cup \mathcal{C}_{join})\}\end{aligned} \tag{4.3}$$

Recall that so far we have constructed three different sets of cores, each possibly being empty. These sets being: the set of cores directly inherited from the ancestor block $\mathcal{C}_{inherited}$, the set of cores directly computed from edge labels $\mathcal{C}_{label}$, and the set of join cores $\mathcal{C}_{join}$. Note that $\mathcal{C}_{label}$ and $\mathcal{C}_{join}$ are mutually exclusive, exactly one of these two sets is empty. Using this extra information we will further explain the construction of $\mathcal{C}_{intersect}$. Note that the construction of intersection cores is split across two main branches, one where exactly one inherited core is used in each intersection, and another branch where no inherited cores are used. Besides this, both branches compute the intersection of all subsets of the union of the label and join core sets, note that order does not matter when computing intersections.

Given the number of core candidates that are generated by this computation we would ideally like to prove that at least they are all actual cores. Unfortunately, this statement is false as we will show next.

Theorem 11. *The intersection of two distinct* CPQ *cores is not always a* CPQ *core itself.*

Proof. This statement can be easily proved by giving a counter example. For example, consider the *CPQ* cores $a \circ b$ and $a \circ (a \cap b)$. Note that the intersection of both *CPQ*s is $(a \circ b) \cap (a \circ (a \cap b))$, which is trivially not a core as the path $a \circ b$ is present twice. This means that the core of this *CPQ* is instead $a \circ (a \cap b)$. Therefore, by contradiction the core of two distinct *CPQ* cores is not necessarily a core itself. □

Clearly this provides a convincing argument that we cannot simply assume the intersection of two *CPQ* cores to be a core itself. However, it is worth investigating why this proof failed. Note that in the given counter example one of the cores is a subgraph of the other core. Clearly this means that any structure present in this core was already present in the other graph. This suggests that the proof may work if we exclude these cases. However, this is unfortunately not true.

Theorem 12. *The intersection of two* CPQ *cores that are not a subgraph of each other is not always a* CPQ *core.*

Proof. Similar to the previous proof we can again easily prove this statement with a counter example. Consider the *CPQ* core $((a^- \circ a) \cap (b^- \circ b)) \circ b^-$, if we intersect this core with the *CPQ* core b^-, the entire *CPQ* collapses down to $((a^- \circ a) \cap id) \circ b^-$ to form the new core. Thus again, by contradiction, even if the *CPQ* cores being intersected are not a subgraph of each other, the result does not need to be a *CPQ* core. □

This failed proof leads to one of the main problems surrounding getting a better runtime for intersection cores. Note that the reason this proof failed is because the source and target designation are not true labels. While a query homomorphism requires the source and target to be mapping to the source and target respectively, the opposite is not true. Note that a regular homomorphism is stricter on this front as it does not allow vertices with a different label to be mapped to each other in any direction. Intuitively, it is easy to think of a *CPQ* as a collection of paths from the source node to the target node. This view is not entirely wrong, but it ignores the fact that some paths may visit the source or target node multiple times on their way from source to target. This in turn means that when adding a new intersecting path to a core, some existing paths may be able to go back past the source to reuse part of this structure before continuing to the target as intended. We have dubbed this problem 'backflow' and it is a major source of unwanted core candidates in the *CPQ*-native Index. Note that we are effectively shortening paths when reusing structure from newly added intersections. However, this also means that these shorter paths are valid existing paths, which in turn means we are already computing them directly via a different route. Therefore, not being able to treat the source and target nodes as true fix points results in a lot of superfluous cores being computed. In fact, note that if the source and target were true fix points, the proof would have worked. Unfortunately, we are currently not aware of any good strategies to avoid this problem, making this problem part of the future work discussed in Section 6.1.9. This in turn means that we cannot assume intersection cores to be true cores and therefore have to always compute their core before adding them to the index.

However, this does not mean that we cannot make use of at least some of the discovered information to improve the performance of intersection core computation. As noted, we know for a fact that the intersection of a core with a core that is a subgraph of that core, results in a *CPQ* that is never a core. Given the exponential nature of the intersection computation approach it is beneficial to at least prevent these *CPQ*s from being considered as core candidates. Fortunately, this is relatively easy to do, since we want to detect if one query graph is a subgraph of another query graph we only need to perform a regular query homomorphism test in both directions.

Note that there will never be a query homomorphism in both directions as that would imply that the *CPQ*s are query homomorphically equivalent, and all the cores computed so far are unique cores. Finally, we simply disallow any subsets with a pair of these one-way homomorphic cores.

To close off this Section, note that Theorem 11 and 12 leave open a second counter example that is not relevant to this section. Since we are ignoring *CPQ*s with top level intersections in this section, we did not consider these as counter examples for the proof. However, note that the intersection of the cores $a \cap b$ and $a \cap c$ results in $(a \cap b) \cap (a \cap c)$, which is trivially not a core as a is listed twice.

4.2.5 Identity Cores

The final set of cores to compute are those cores involving identity. Note that this step is only required if the block we are computing cores for actually stores paths that are loops. If a block does not store looping paths, then it can never have a top level intersection with identity. Actually computing identity cores is relatively straightforward, the general equation for this procedure is shown in Equation 4.4

$$\mathcal{C}_{identity} = \{q \cap id \mid q \in (\mathcal{C}_{join} \cup \mathcal{C}_{label} \cup \mathcal{C}_{intersect})\} \tag{4.4}$$

Essentially, we are taking all the cores we have computed and adding their intersection with identity as a potential core candidate. Note that we ignore inherited cores here, as they have already been intersected with identity in their original block. Next we will again show that these core candidates are unfortunately not always cores.

Theorem 13. *The intersection of a* CPQ *core and identity is not always a core.*

Proof. We will prove this by contradiction. Consider the *CPQ* core $a \cap a^-$, this is a relatively common pattern where the same edge is present between two nodes in both directions. However, this also means that these edges can only be distinguished due to the source and target node being two distinct vertices. Adding an intersection with identity results in $(a \cap a^-) \cap id$, which is essentially the same graph, except the source and target vertex have been merged into the same vertex. This makes it so there are two a loops on this vertex, while the core of this vertex would only have one. □

Essentially, this means that cores that have the same structure after the source and target vertex are swapped are not cores after being intersected with identity. In general, intersecting a core with identity has a reasonably high chance to result in a *CPQ* that is no longer a core. As such, also in this case we always need to compute the core explicitly before adding the result to the index.

Finally, there is one potential optimisation that was not implemented. Note that some of the inherited cores will already have been intersected with identity. This also means that all the cores these cores were intersected with during the intersection step from Section 4.2.4 will be loops already. Thus intersecting these cores with identity could be skipped. The primary reason this optimisation was not implemented is because not all blocks have identity cores, so other optimisations were prioritised.

4.3 Completing the Index

Having introduced all the different types of cores that need to be computed, we can now put together a visualisation of the complete *CPQ*-native Index for the example graph with cores. This visualisation is shown in Figure 4.5 and again also includes the index data for $k = 1$ for more clarity, note that this figure is essentially an augmented version of Figure 4.3. The arrows again indicate the ancestor relationship between nodes, with ancestor blocks pointing to the blocks that were refined from it. Note that all shown cores are actually stored as the canonical binary forms covered in Section 3.4.3.2. Since this format is not human readable, we instead use a single representative *CPQ* from the query homomorphic equivalence class instead.

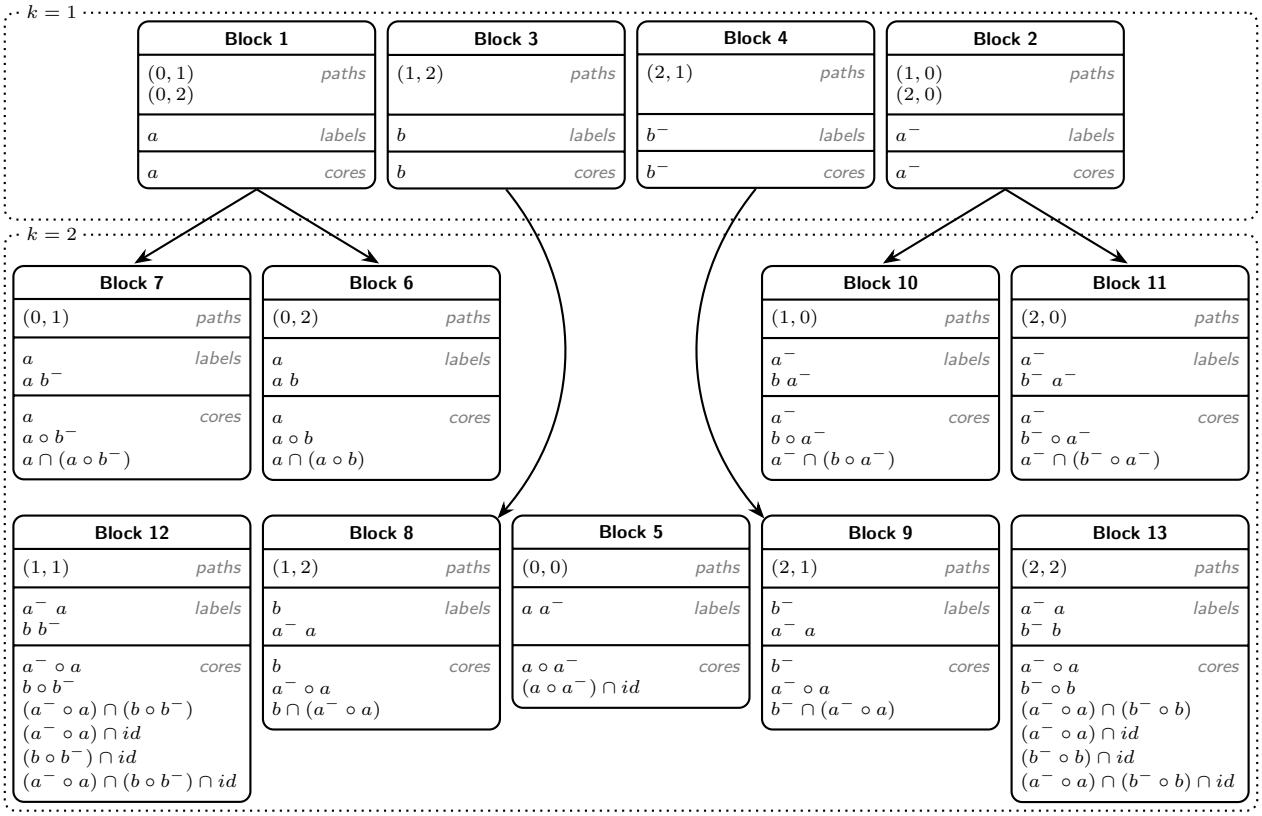


Figure 4.5: The fully constructed *CPQ*-native Index with cores.

Although this visual representation of the index works well for illustrative purposes, it is not the form of the index the computer will work with. To fully complete the index we will construct two maps:

1) The first map $\mathcal{I}_{c2b}$ maps from canonical *CPQ* core to a set of block identifiers. Recall that the original language-aware index [65] used label sequences as the search key. Thus, this map is a replacement for the map from label sequences to block identifiers used in the language-aware index, since the *CPQ*-native Index uses canonical cores as search keys instead.

2) The second map $\mathcal{I}_{b2p}$ maps from block identifier to a set of stored paths. This map is identical to the other map used in the language-aware index, since we are still partitioning all the paths in the graph by k-path-bisimulation.

Together these two maps from the *CPQ*-native Index. Both maps can be derived from the visual representation of the index in Figure 4.5 with relative ease, the first map $\mathcal{I}_{c2b}$ from cores to blocks is shown in Equation 4.5. Note that for clarity plain *CPQ*s are again used as the keys instead of their binary canonical form.

$$
\begin{aligned}
a &\mapsto \{6,7\}\\
a \circ a^- &\mapsto \{5\}\\
a \circ b &\mapsto \{6\}\\
a \circ b^- &\mapsto \{7\}\\
a \cap (a \circ b) &\mapsto \{6\}\\
a \cap (a \circ b^-) &\mapsto \{7\}\\
a^- &\mapsto \{10,11\}\\
a^- \cap (b \circ a^-) &\mapsto \{10\}\\
a^- \cap (b^- \circ a^-) &\mapsto \{11\}\\
a^- \circ a &\mapsto \{8,9,12,13\}\\
b &\mapsto \{8\}\\
b \cap (a^- \circ a) &\mapsto \{8\}\\
b \circ a^- &\mapsto \{10\}\\
b \circ b^- &\mapsto \{12\}\\
b^- &\mapsto \{9\}\\
b^- \cap (a^- \circ a) &\mapsto \{9\}\\
b^- \circ a^- &\mapsto \{11\}\\
b^- \circ b &\mapsto \{13\}\\
(a \circ a^-) \cap id &\mapsto \{5\}\\
(a^- \circ a) \cap (b \circ b^-) &\mapsto \{12\}\\
(a^- \circ a) \cap (b \circ b^-) \cap id &\mapsto \{12\}\\
(a^- \circ a) \cap (b^- \circ b) &\mapsto \{13\}\\
(a^- \circ a) \cap (b^- \circ b) \cap id &\mapsto \{13\}\\
(a^- \circ a) \cap id &\mapsto \{12,13\}\\
(b \circ b^-) \cap id &\mapsto \{12\}\\
(b^- \circ b) \cap id &\mapsto \{13\}
\end{aligned}
\tag{4.5}
$$

Also note that this representation serves as a form of compression for the cores stored in the index. The original index as a collection of distinct blocks in Figure 4.5 contains 32 cores, while the map shown in Equation 4.5 only contains 26 entries. This compression effect is more significant for larger graphs as cores are more likely to be shared between more blocks. The second map from blocks to paths $\mathcal{I}_{b2p}$ is shown in Equation 4.6. Note that this map is relatively simple as each block only contains a single path

$$
\begin{aligned}
5 &\mapsto \{(0,0)\}\\
6 &\mapsto \{(0,2)\}\\
7 &\mapsto \{(0,1)\}\\
8 &\mapsto \{(1,2)\}\\
9 &\mapsto \{(2,1)\}\\
10 &\mapsto \{(1,0)\}\\
11 &\mapsto \{(2,0)\}\\
12 &\mapsto \{(1,1)\}\\
13 &\mapsto \{(2,2)\}
\end{aligned}
\tag{4.6}
$$

Finally, we will briefly comment on the correctness of the completed index in terms of its completeness. Our focus here is on collecting all of the arguments presented throughout Section 4.2 in one place. The main question of interest here is if there are any cores that are missing from the completed index. Trivially, and perhaps surprisingly, the answer here is yes. The core for only identity *id* is intentionally missing from the completed index. Note that if present it would have had to be stored as a core for block 5, 12 and 13 in Figure 4.5. Instead, we deliberately chose to exclude this core from the index to prevent it from participating in the

join step discussed in Section 4.2.3 any more than it already does. This because, as mentioned, joining any *CPQ* with identity does nothing, so this is just wasted computation time. To a lesser extent it would also affect the optimisation for the intersections step in Section 4.2.4, as the identity *CPQ* is a subgraph of any *CPQ* and therefore also query homomorphic to any *CPQ*, resulting in more wasted computation time. This also means that including the identity *CPQ* would not negate the need for the identity step discussed in Section 4.2.5. Therefore, given its problems we instead chose to handle the identity query separately, as will be discussed in Section 4.4. Note that it could have been integrated in the current computation flow, we simply did not choose this option. Going back to our original completeness question, this means we need to slightly alter our statement to exclude this case.

Theorem 14. *The presented core computation steps correctly compute all CPQ_k cores for a graph except for 'id'.*

Proof. Note that the correctness of our approach essentially stems from the fact that we are closely following every available construction step in the *CPQ* grammar introduced Equation 2.1 in Section 2.2.2 during core computation.

Starting with our base case. In the case that our initial set of cores is just all edges with a specific label we can technically refer to the language-aware indexing paper [65] for correctness. However, note that this special case is really just a search for all edges with some label. Trivially, if we collect all edges with some label we have all edges with this label.

The first regular step of the computation procedure is the computation of join cores. The goal of this step is to compute all members of CPQ_k that map to a certain block with a diameter of exactly k and no top level intersections. However, note that this step actually also generates a number of other core candidates we do not really need, though this does not affect correctness. Note that the correctness of the core goal of this step essentially stems from the approach in the original language-aware indexing paper [65]. Instead of directly computing new label sequences when combining blocks, we save these combinations for later. Moreover, we save combinations of blocks of any length that together sum to k. Under the assumption that these blocks correctly have all their cores computed, this means that the join of all the cores in these blocks does generate all cores of length k without top level intersections. If we assume that two *CPQ* cores do exist that can be combined to form a new core, then this raises the question of why the respective blocks for these cores were not combined. Note that this again stems from the correctness of the partitioning approach based on k-path-bisimulation, which is discussed in more detail in the original papers on this topic [65, 24, 25].

Moving on to intersections, note that the core goal when computing intersection cores is computing all top level intersections of any number of members in CPQ_k that have a diameter of exactly k. The only real factor in the correctness of this step is the assumption that the cores computed so far are the complete set of all CPQ_k cores without top level intersections, which follows from the rest of this proof.

Finally, the identity cores. This step is similarly very straightforward as we are essentially just adding an intersection with identity to any core we computed. If any cores with an intersection with identity were missed this would imply that the rest of the procedure is incorrect.

These arguments combined with existing proofs in the literature confirm that we have computed all cores, though as noted, we are computing more cores than strictly necessary. □

Next we will briefly touch on the algorithmic complexity of index construction. Note that partitioning the graph according to k-path-bisimulating is essentially unchanged from the language-aware indexing paper [65]. As mentioned, we only add a factor k to the runtime, this means this part of the runtime remains polynomial. For the core computation, note that as mentioned in Section 4.2.4 the runtime is dominated by the intersection cores step. From this step we know that an exponential number of core candidates has to be computed. We also know that the canonical core computation performed for each of these candidates takes polynomial time [26]. However, this fact is less relevant given that the exponential number of cores already makes it so the final runtime is exponential. Nevertheless, it is worth noting that this runtime is in the number of cores present in a single block. Exactly how many cores end up in a specific block depends on the exact input graph and as we will see in Chapter 5 this distribution is highly uneven.

4.4 Querying the Index

Having finally constructed the entire *CPQ*-native Index, we are now at the point where we can use it to answer queries. All the constructs required for this process have already been introduced in Section 4.3, specifically the maps $\mathcal{I}_{c2b}$ and $\mathcal{I}_{b2p}$ and their example graph realisations in Equations 4.5 and 4.6 respectively. Using these structures we will show how to run the query $a^- \circ a$ on the example graph from Figure 4.1. However, before covering the regular querying process, we will show how to handle the query of just identity.

As mentioned in Section 4.3, the core of the identity query was intentionally excluded from the index. The best way to handle this query depends on the exact deployment of the index. Note that this query effectively just selects all the vertices in the graph. As such, the most intuitive solution is to simply let the existing infrastructure of the graph database handle this query instead, as it is likely that a graph database has some efficient mechanism for fetching all vertices in the graph. There is no easy way for the *CPQ*-native Index to store this information without building exactly such an index itself, which would generally most likely be redundant. The current reference implementation employs a slightly lacking implementation that does not use up extra storage, but does sometimes miss results due to this. The solution used by the current index implementation is to simply scan all blocks for blocks containing only looping paths, these paths are then all returned as the answer. Note that this solution is lacking because it requires that vertices have edges attached to them. Moreover, this technique only starts to work well for an index with at least $k = 2$, since it is likely that many vertices do not have a path of length one leading to them. Note that at length two any vertex with at least one edge will be found as the query that follows this edge and then follows it back again will match the vertex. At this point, the only vertices that will be missed are vertices without any edges. Although as mentioned it would be easy to just keep a list of all vertices in the graph, this would often be redundant and identity is not a query we expect to see often in practice either. Thus for the current implementation this alternative was adopted.

Next we will discuss the more general and interesting case for most queries. Essentially, the general flow when evaluating a query $q \in CPQ$ is to perform the following steps:

1) Check if $\mathrm{dia}(q) > k$, we cannot answer queries with a diameter larger than what the index was constructed for. These queries will be rejected by the index and should be processed differently by the database or broken down into smaller queries as discussed in Section 6.1.3.

2) Check if $\mathrm{dia}(q) = 0$, this means the query is *id*, as mentioned this query has some form of special handling.

3) Compute the query graph $\mathcal{G}_q$ of q using the functions $f_{q2graph}$ and f_{merge} introduced in Section 2.2.3.

4) Compute the core of $\mathcal{G}_q$ using the algorithms and techniques discussed in Section 3.2 or Section 3.3.

5) Compute the binary canonical representation of the core using the methodology discussed in Section 3.4.

6) Look up the relevant block identifiers using the canonical core in $\mathcal{I}_{c2b}$.

7) Retrieve the relevant paths using the block identifiers from $\mathcal{I}_{b2p}$.

8) Combine all the retrieved paths into a single set and return them as the evaluation result.

Most of these steps are relatively straightforward and have already been covered extensively in different parts of this thesis. Thus we will focus on the last three steps to reiterate some important facts in order to explain why this approach is correct. Recall from Section 4.1 that the graph was partitioned into disjoint equivalence classes where paths within the same block cannot be distinguished by any CPQ_k. In other words, when evaluating a query either all or no paths in a block need to be returned. This also explains why we can safely combine the paths in all the returned blocks, as no paths are shared between these blocks since the blocks are all disjoint, so there will be no overlap in the collected paths. Finally, we give a formal characterisation of the query evaluation process as f_{eval} in Equation 4.7.

$$\begin{aligned} \mathcal{G}_q &= f_{merge}(f_{q2graph}(q, v_s, v_t, (\{v_s, v_t\}, \emptyset, \mathcal{L}, v_s, v_t, \emptyset))) \\ f_{eval}(q) &= \bigcup_{B \in \mathcal{I}_{c2b}(\mathrm{canon}(\mathrm{core}(\mathcal{G}_q)))} \left(\bigcup_{b \in B} \mathcal{I}_{b2p}(b) \right) \end{aligned} \tag{4.7}$$

Next we will show a concrete example by evaluating the *CPQ* $a^- \circ a$ on the example graph from Figure 4.1. For this we will again act as if our index keys are plain *CPQ*s instead of binary canonical forms. First we will evaluate $\mathcal{I}_{c2b}$, recall that this map is listed in Equation 4.5 in Section 4.3. The result of this evaluation is shown in Equation 4.8.

$$\mathcal{I}_{c2b}(a^- \circ a) = \{8, 9, 12, 13\} \tag{4.8}$$

Next we will evaluate $\mathcal{I}_{b2c}$ for each of the retrieved blocks, recall that this map is listed in Equation 4.6 in Section 4.3. The result of each of these retrievals is shown in Equation 4.9.

$$\begin{aligned} \mathcal{I}_{b2p}(8) &= \{(1,2)\} \\ \mathcal{I}_{b2p}(9) &= \{(2,1)\} \\ \mathcal{I}_{b2p}(12) &= \{(1,1)\} \\ \mathcal{I}_{b2p}(13) &= \{(2,2)\} \end{aligned} \tag{4.9}$$

Finally, we combine these results to form the complete answer to the original query as shown in Equation 4.10

$$f_{eval}(a^- \circ a) = \{(1,2)\} \cup \{(2,1)\} \cup \{(1,1)\} \cup \{(2,2)\} = \{(1,2),(2,1),(1,1),(2,2)\} \tag{4.10}$$

4.5 Limiting Intersections

As already briefly discussed, most of the memory and time required to construct the index is used by the intersection cores introduced in Section 4.2.4. While it could reasonably be expected that a user may want to query for the intersection of two interesting *CPQ*s, it is unlikely that a user would be querying the intersection of 7000 *CPQ*s very frequently, if at all. Currently, the index stores any intersection of any number of cores. If there is no chance that these cores will actually ever be used, then it makes more sense to exclude them from the index. Note that this is effectively a small step towards an interest-aware *CPQ*-native Index where the user decides which cores are worth storing. Limiting the number of cores by limiting intersections is also extremely effective at reducing the time and memory required to construct an index for a graph. Consequently, introducing a limit like this makes it possible to compute a *CPQ*-native Index for larger datasets, further expanding its applicability to various use cases. Next we will shows how to introduce such a limit into the existing procedure.

Essentially, the idea is to limit the subsets step from Section 4.2.4 to allow at most a predefined number of intersections i. Following this we can update the original computation from Equation 4.3 to the new version shown in Equation 4.11.

$$\begin{aligned} \mathcal{C}_{intersect} = &\{q_1 \cap \cdots \cap q_n \mid \{q_1, \ldots, q_n\} \subseteq (\mathcal{C}_{label} \cup \mathcal{C}_{join}) \wedge n \geq 2 \wedge n \leq i\} \\ &\cup \{q_0 \cap q_1 \cap \cdots \cap q_n \mid q_0 \in \mathcal{C}_{inherit} \wedge \{q_1, \ldots, q_n\} \subseteq (\mathcal{C}_{label} \cup \mathcal{C}_{join}) \wedge n < i\} \end{aligned} \tag{4.11}$$

Note that enforcing the limit in this way means that we do not count a potential intersection with identity towards the limit, this decision is intentional as in graph form intersecting with identity does not actually increase the number of paths between source and target.

4.6 Summary

In this chapter we have presented all the logic required to construct a *CPQ*-native Index for a graph. The presented construction is split up into two main parts. The first part is a slightly modified version of the k-path-bisimulation based partitioning logic from the original language-aware index [65], and in the second part we have introduced numerous novel techniques required to use *CPQ*s as the keys to the partition blocks created in the first part. Using these techniques we have also successfully answered our second sub-question from Section 1.2 on how to associate cores with index blocks. Finally, we have also discussed how to evaluate queries using the fully constructed index and how to support a limited form of interest awareness. Next we will evaluate the performance of the presented index in Chapter 5.

Performance Evaluation

5 第五章

Now that the *CPQ*-native Index has fully been introduced, we will investigate its performance on actual datasets. In this chapter we will first look at the performance of the *CPQ* core computation algorithm in Section 5.1. After the core algorithm we will investigate the performance of the complete index in Section 5.2. For the performance evaluation we will make use of the 'Garuda' server from Osaka University. This is a HPE ProLiant DL385 Gen10 Plus server running Ubuntu Linux 18.04 LTS with 2048GB of DDR4 RAM and two AMD EPYC 7542 CPUs with each 32 cores for 64 cores total, each running at between 2.9GHz and 3.4GHz. The Java version used for all tests was Eclipse Temurin Java 17 LTS [21].

5.1 Core Computation

Since a major part of Chapter 3 focusses on designing and optimising the core computation algorithm we will briefly discuss the performance of this algorithm before discussing the performance of the index. Recall that we designed two algorithms for *CPQ* core computation, one naive algorithm in Section 3.2 and one efficient algorithm in Section 3.3. In this section we will compare the average performance of both algorithms against each other on datasets containing random *CPQ* query graphs of varying sizes. For this comparison we will be using gMark at commit `ee76a39e` [38] for the naive core algorithm and gMark version 1.2 [37] for the improved algorithm. In order to properly make use of all the cores available in our server we will multi-thread the core computations.

5.1.1 Datasets

The datasets we will be using to evaluate the core algorithms each consist of 1024 random *CPQ* query graphs. These datasets were all generated by running the *CPQ* query graph construction procedure from Section 2.2.3 on random *CPQ*s generated by the plain *CPQ* generation algorithm developed during the CPQ Keys project [35]. Although this *CPQ* generation algorithm can be found in Algorithm 1 in Section 2.5.1 of the original report, it has also been included in this thesis as Appendix A. The size of these datasets is determined by the rules parameter of the algorithm and the name of the dataset matches the value for the rules parameter that was used to generate it. All of the datasets were generated by gMark version 1.2 [37] with 1234 as the random seed. An overview of the average size of the graphs in each dataset with standard deviation is given in Table 5.1. The sizes from this table are also plotted in Figure 5.1 with error bars indicating the standard deviation. This plot reveals linear graph size scaling and relatively little variation in graph size within each dataset.

Table 5.1: Overview of dataset sizes.

Dataset	Graphs	Vertices	Edges
4	1024	3.51 ± 1.34	4.18 ± 0.87
8	1024	5.07 ± 1.80	7.38 ± 1.39
16	1024	8.21 ± 2.50	13.71 ± 2.25
32	1024	14.53 ± 3.66	25.98 ± 3.81
64	1024	27.09 ± 4.77	50.57 ± 5.60
128	1024	52.56 ± 6.83	99.07 ± 8.81
256	1024	103.11 ± 10.03	195.04 ± 13.31
512	1024	203.96 ± 14.02	385.34 ± 20.20

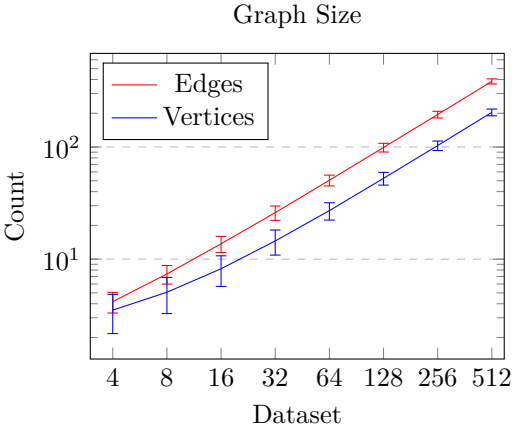


Figure 5.1: Dataset size scaling.

5.1.2 Core Computation Performance

After running both core computation algorithms on the datasets, we obtain the results in Table 5.2. The naive algorithm did not finish the last dataset within the allotted time. For each algorithm the fastest and slowest core computation time was recorded, as well as the average core computation time together with the standard deviation. Note that the standard deviation is extremely large, in particular for the naive algorithm. The difference between the minimum and maximum core computation time for each algorithm further highlights the extreme variance. However, it is worth noting that the runtime variation for the improved algorithm is significantly lower than that of the naive algorithm. Nevertheless, as can be seen for dataset 64 and 512, the improved algorithm still has large outliers. Unfortunately, the presence of outliers like this is to be expected given the quadratic and exponential nature of some parts of the algorithm. In fact, the variance appears to be lower than expected, meaning the filtering operations are generally able to keep the size of the intermediate results in check.

Table 5.2: Runtimes for both core computation algorithms on the test datasets.

	Improved Algorithm			**Naive Algorithm**		
Dataset	**Min**	**Max**	**Average**	**Min**	**Max**	**Average**
4	68 µs	13 ms	2 ms ± 2 ms	11 µs	37 ms	3 ms ± 4 ms
8	209 µs	28 ms	4 ms ± 3 ms	6 µs	468 ms	30 ms ± 55 ms
16	282 µs	49 ms	8 ms ± 4 ms	114 µs	91 ms	9 ms ± 15 ms
32	826 µs	55 ms	9 ms ± 6 ms	417 µs	546 ms	47 ms ± 57 ms
64	524 µs	157 ms	14 ms ± 27 ms	8 ms	10 s	918 ms ± 1 s
128	3 ms	43 ms	13 ms ± 6 ms	451 ms	4 min 40 s	38 s ± 35 s
256	3 ms	43 ms	13 ms ± 6 ms	48 s	2 h 8 min 24 s	28 min 12 s ± 20 min 3 s
512	223 ms	31 min 54 s	4 s ± 1 min 11 s	-	-	-

The results from Table 5.2 have also been visualised in Figure 5.2. The solid lines in this plot represent the average runtime, while the shaded regions show the area between the minimum and maximum runtime. Note that we do not use the standard deviation to show variation in runtime as it is too large to give useful information, resulting in negative lower bounds for a number of data points. Although hard to say for sure, both plots show remarkably linear runtime scaling for large sections, though the runtime for both algorithms ultimately appears to have at least a quadratic trend, presumably caused by outliers. However, note that even though both plots appear close, there is still a large difference in actual runtime due to the double log scale used in this plot.

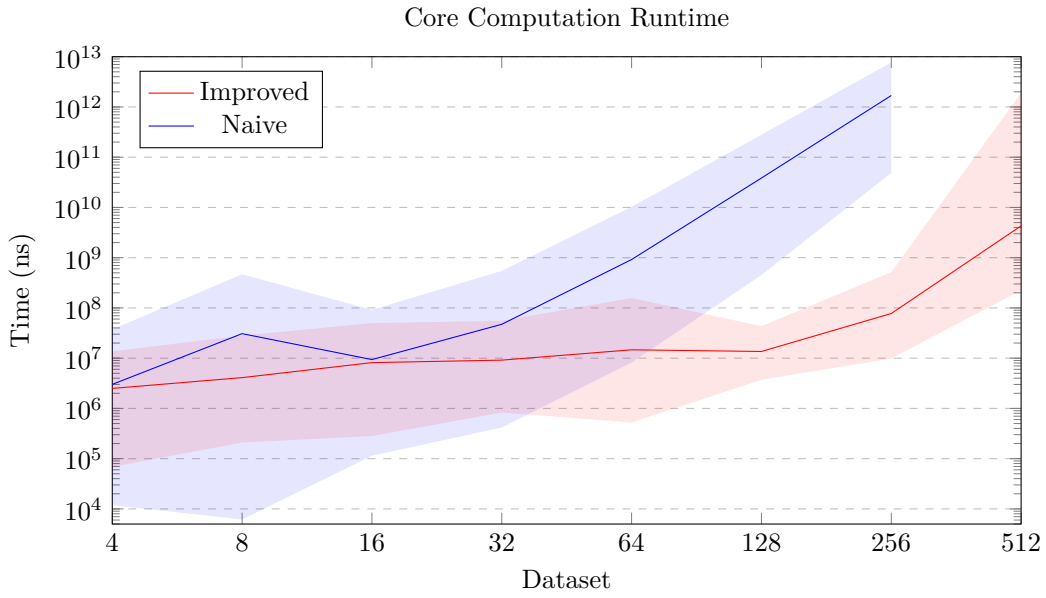


Figure 5.2: Runtime scaling for both *CPQ* core computation algorithms.

To conclude, it is clear that the improved algorithm is several orders of magnitude faster than the naive algorithm. Moreover, on average the improved algorithm is also faster for small graphs even though it has to track more data. However, it is worth noting that on specific small graph cases the naive algorithm can be

faster. Nevertheless, the improved algorithm is the clear victor for *CPQ* core computation. Finally, it is worth noting that there may be a certain bias in the results presented here. As noted in Chapter 3, the homomorphism test is not strictly limited to query homomorphism, meaning it could be employed to compute different types of graph cores. Since the dataset is composed entirely of *CPQ* query graphs, we do not know if the structure of these graphs potentially favours the improved algorithm. For our use case this is irrelevant, but if the algorithm is used in a more general setting additional tests should be conducted.

5.2 Index Computation

Next we will investigate the performance of *CPQ*-native Index itself. For this we will first briefly look at the partitioning performance in Section 5.2.2 and then discuss the performance of the core computation in more detail in Section 5.2.3. All of the collected data was obtained by using gMark version 1.2 [37] and version 1.0 [36] of the *CPQ*-native Index. Note that the implementation of the *CPQ*-native Index distributes core computation by index block. This means that cores for multiple blocks can be computed at the same time by different CPU cores. However, it is not possible for multiple CPU cores to work together on the same index block.

5.2.1 Datasets

We will start by introducing the graph datasets we will be using to evaluate the Index. Five of the datasets we will use are the same as the datasets used in the language-aware indexing paper [65], these datasets are: 'Robots' [69, 67], 'Advogato' [45], 'BioGrid' [69, 8], 'ego-Facebook' [66, 65], and 'Epinions' [45, 59]. The last dataset called 'Example' is an extremely small graph which we will use to run our tests that do not limit intersections, this dataset will be discussed in more detail in Section 5.2.1.1. All the datasets and their sizes are tabulated in Table 5.3, the sizes of the graphs are also plotted in Figure 5.3. Note that two of the datasets were augmented with synthetic edge labels for the language-aware index paper, as these graphs were originally unlabelled.

Table 5.3: Overview of dataset sizes.

Dataset	Vertices	Edges	Labels	Real
Example	14	27	2	Yes
Robots	1484	2960	4	Yes
ego-Facebook	4039	88 234	8	No
Advogato	5417	51 327	4	Yes
BioGrid	64 332	862 277	7	Yes
Epinions	131 828	840 799	8	No

Note: 'Real' indicates if the edge labels are real or synthetic.

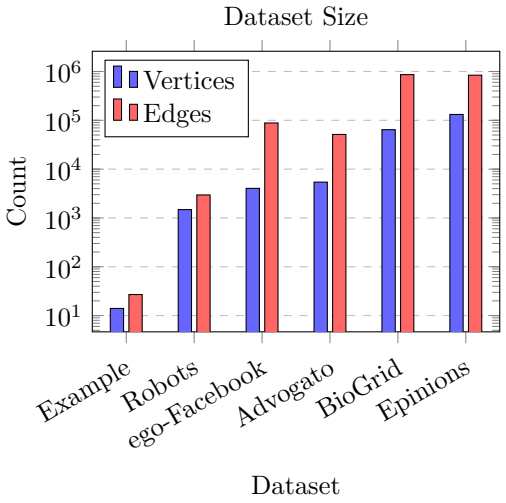


Figure 5.3: Dataset size comparison.

Note that the number of labels and edges for each dataset are halved compared to the language-aware indexing paper. This is due to the fact that we can natively deal with reverse edges and thus do not need to consider forward and backward edges as separate constructs. This means that we did not need to introduce new distinct labels to act as inverted labels and consequently also did not need to duplicate all edges to add a reverse variant. Finally, note that the datasets are currently sorted based on vertex count. However, some of the datasets actually have more edges than the datasets after them in this order.

5.2.1.1 Example Graph

As mentioned in Section 5.2.1, we will make use of an extremely small graph for testing the index without intersection limit. This graph was modelled after a small social media network and is a relabelled but structurally identical version of the example graph given in Figure 1 of the language-aware indexing paper [65]. The entire graph is visualised in Figure 5.4.

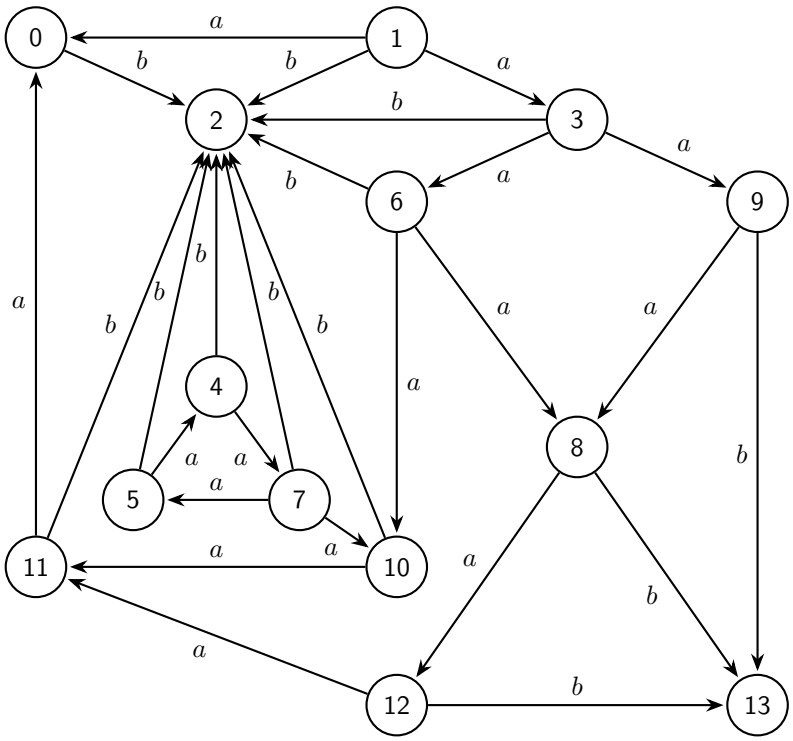


Figure 5.4: The example graph dataset.

5.2.2 Partitioning

Although we will include some results here on the time it takes to construct the partition blocks for the index, it should be stressed that there is very little meaningful information in this data. The implementation for the original language-aware index [65] has a number of optimisations for partitioning that were not implemented in the reference implementation for this thesis as optimising core computation was given priority. Although we do need to track more information for core computation, meaning a higher runtime is expected, it is likely that these optimisations could significantly improve the current runtimes presented in this section. Moreover, the partition blocks we are computing here save additional information required to compute cores later on. This means that the index size numbers are significantly inflated and in some cases even larger than the indices with cores we will compute in Section 5.2.3. Nevertheless, in Table 5.4 we list for the various datasets from Section 5.2.1 and varying diameters k, the time required to partition the paths according to k-path-bisimulation, the time required to combine k-path-bisimilation paths into blocks, and the saved index size. Note that the total edge count of a graph appears to have a larger impact on the partitioning time than the total vertex count.

Table 5.4: Overview of index graph partitioning times.

Dataset	k	**Partitioning**	**Block Construction**	**Saved Size**
Example	1	605 ms	460 ms	613 B
Example	2	888 ms	352 ms	2.9 KiB
Example	3	1 s	319 ms	16.1 KiB
Example	4	1 s	217 ms	45.0 KiB
Example	5	2 s	238 ms	93.7 KiB
Example	6	2 s	511 ms	163.6 KiB
Example	7	3 s	238 ms	255.1 KiB
Example	8	3 s	249 ms	368.3 KiB
Robots	2	691 ms	115 ms	1.9 MiB
Robots	3	1 min 7 s	973 ms	21.7 MiB
ego-Facebook	2	19 min 27 s	3 s	111.0 MiB
Advogato	2	2 min 56 s	1 s	56.7 MiB
BioGrid	2	27 h 41 min 40 s	1 min 4 s	1.3 GiB
Epinions	2	19 h 1 min 35 s	2 min 6 s	2.6 GiB

5.2.3 Cores

Having constructed several base indices without cores in Section 5.2.2, we will now attempt to compute cores for the indices. Recall that we have two main parameters we can vary when computing cores. The first parameter k is related to partitioning and controls the diameter of the index. Consequently, we will never be able to compute *CPQ* cores of a diameter larger than k. The second parameter i controls the maximum number of intersections and was introduced in Section 4.5. By varying both of these parameters several *CPQ*-native indices with cores were computed, and the computation times and other statistics are shown in Table 5.5.

Table 5.5: Overview of index core computation times.

Dataset	k	i	**Cores**	**Mapping**	**Total Cores**	**Unique Cores**	**Saved Size**
Example	1	∞	1 s	293 ms	4	4	555 B
Example	2	∞	1 s	484 ms	100	42	2.5 KiB
Example	3	∞	18 min 41 s	3 s	8 092 559	2 787 169	351.7 MiB
Robots	2	2	2 s	319 ms	191 536	16 110	2.6 MiB
Robots	2	4	42 s	1 s	2 874 994	1 549 353	64.2 MiB
Robots	2	8	25 h 33 min 57 s	9 min 6 s	1 165 832 385	1 049 965 927	65.6 GiB
Robots	3	1	7 s	650 ms	1 419 926	1095	12.6 MiB
ego-Facebook	2	2	2 min 14 s	31 s	147 882 110	32 010	591.4 MiB
ego-Facebook	2	3	21 min 32 s	28 min 1 s	1 949 847 997	2 239 420	7.3 GiB
Advogato	2	2	45 s	9 s	35 595 437	126 080	170.4 MiB
Advogato	2	4	27 min 9 s	24 min 32 s	1 581 466 667	203 922 146	13.2 GiB
BioGrid	2	2	21 min 20 s	10 min 51 s	1 284 712 626	1 875 817	5.5 GiB
Epinions	2	2	1 h 59 min 21 s	1 h 45 min 38 s	7 056 863 666	12 439 681	27.9 GiB

Next we will briefly introduce each of the columns in this table:

1) **Dataset**: The graph dataset from Section 5.2.1 the index was constructed for.

2) k: The *CPQ* diameter as introduced in Section 2.2.2.

3) i: The maximum number of intersections as described in Section 4.5.

4) **Cores**: The total time it took to compute all *CPQ* cores, this is the procedure described in Section 4.2.

5) **Mapping**: The total time it took to construct the map from *CPQ* cores to index blocks, this is the procedure described in Section 4.3.

6) **Total Cores**: The total number of real cores computed for the index, this is the sum of the cores in each block. Note that this total does not include computed candidates that were determined to be duplicates of cores that were already computed and also does not include candidates that were found to not be actual cores.

7) **Unique Cores**: The total number of unique cores in the constructed index, this is the number of keys in the final map.

8) **Saved Size**: Size of the index when saved to disk, note that the index will require more memory when loaded in RAM.

In addition to the runs recorded in Table 5.5, the following runs were attempted but aborted due to running into memory or time limits:

1) 'Example' with $k = 4$ and no intersection limit.

2) 'Biogrid' with $k = 2$ and max intersections 3.

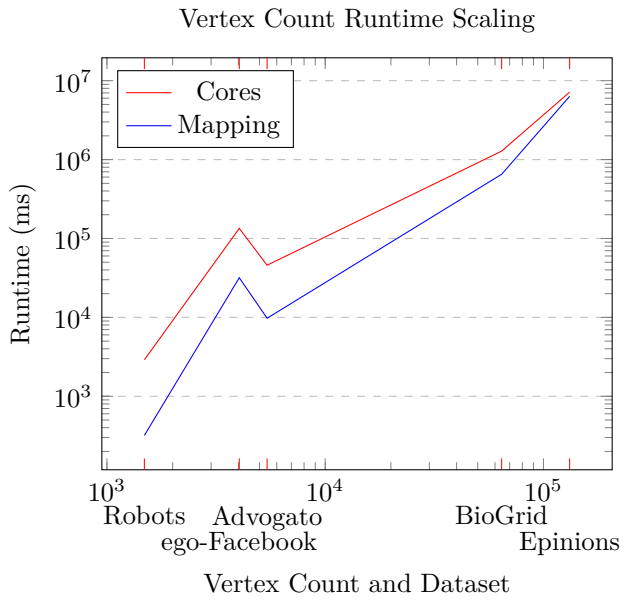
3) 'ego-Facebook' with $k = 2$ and max intersections 4.

4) 'Robots' with $k = 3$ and max intersections 2.

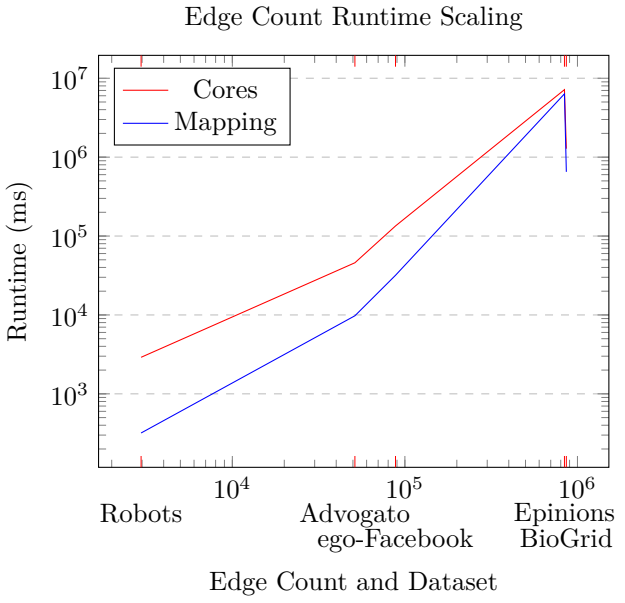
Note that this already implies that creating an index for a larger diameter or with a large intersection limit appears infeasible. However, as we will cover in Section 5.3, this may not present a usability issue. Moreover, the interest unaware version of the language-aware index was also not constructed for a value larger than $k = 2$ in the original paper [65]. In fact, it is worth noting that computing cores with an intersection limit of 2 is frequently faster than computing the partitions. This suggests that at least some level of native *CPQ* handling can be supported without significant computation overhead when memory is not a major concern. In Sections 5.2.3.1, 5.2.3.3 and 5.2.3.2 we will try to develop a better understanding of the core computation process.

5.2.3.1 General Runtime Scaling

Although we have limited data available and only a small collection of datasets, we will start by looking at some general runtime scaling. Since the only configuration we were able to apply to all datasets is $k = 2$ and $i = 2$, this is the data we will visualise. In Figure 5.5 runtime scaling for both the core computation and mapping step is plotted against the number of vertices and edges in the datasets.



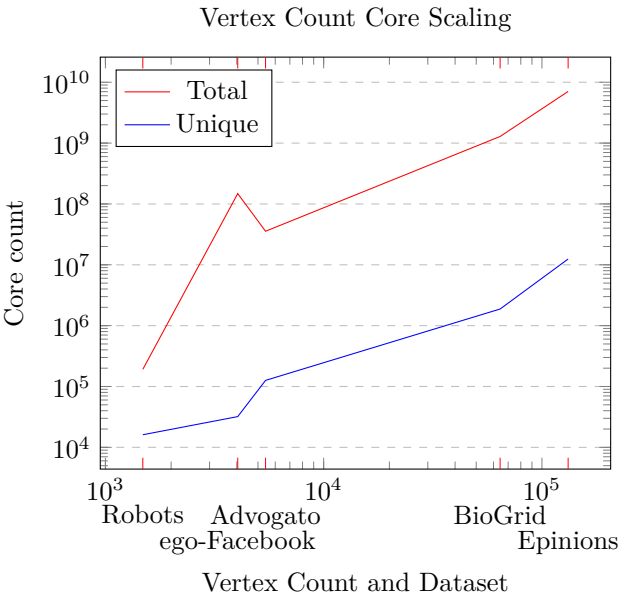
(a) Vertex count runtime scaling.



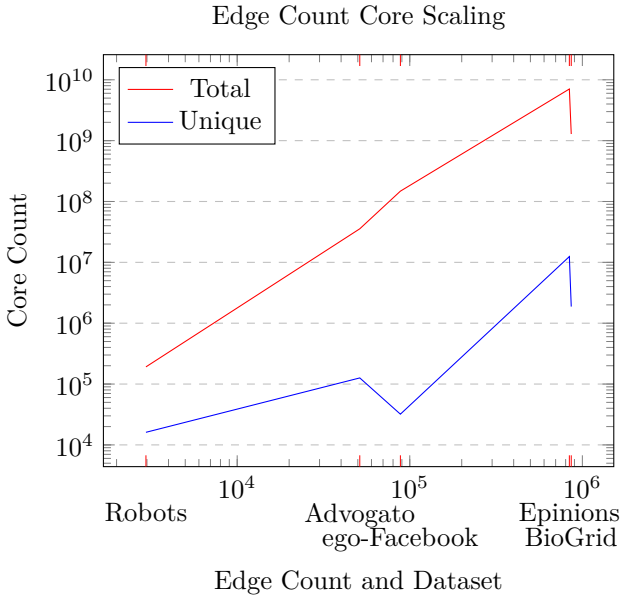
(b) Edge count runtime scaling.

Figure 5.5: Runtime scaling for each dataset for $k = 2$ and $i = 2$.

From these plots it appears that the runtime generally appears to scale linearly in both the number of vertices and the number of edges. However, it is also clear that the dataset itself influences the runtime to a certain extent as neither of the plots is consistently increasing. For our specific sample this can be explained by the edge or vertex count being significantly different, for example for Figure 5.5a 'ego-Facebook' has significantly more edges than 'Advogato', which presumably causes the runtime spike. However, without more data points it is hard to say if this reasoning holds in general or if graph structure also plays a role. In Figure 5.6 we show a similar plot for the core count instead of the runtime.



(a) Vertex count core scaling.

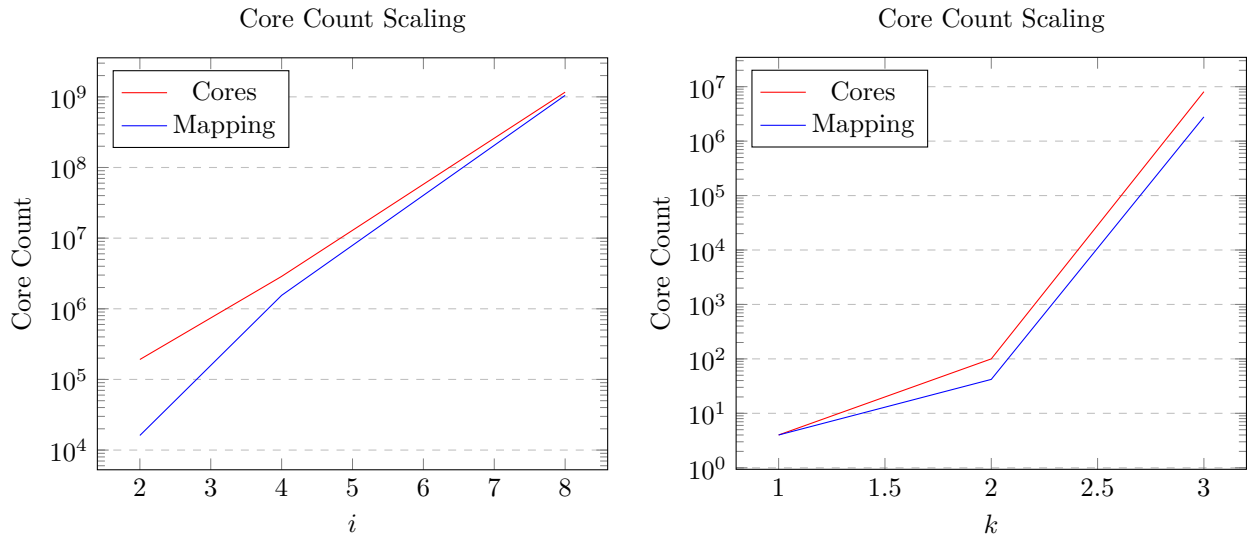


(b) Edge count core scaling.

Figure 5.6: Core count scaling for each dataset for $k = 2$ and $i = 2$.

These plots again show what appears to be a general linear trend with significant distortions. However, the spikes in Figure 5.6a and 5.6b are caused by 'ego-Facebook' and 'Epinions' respectively. These datasets both have the highest label count out of any of the datasets. Since having more labels leads to more distinguishable graph structure, this would also lead to more cores, likely explaining the spikes. However, it is also worth noting that these two datasets are the only two datasets with synthetic edge labels, which may offer an alternative explanation. It is also still important to note that we really do not have enough datasets to say anything conclusive about these relatively minor fluctuations. However, the general trend appearing to be linear instead of quadratic or worse is promising.

Finally, we will look at some scaling properties when varying the diameter and intersection limit. Specifically, we will look at the core count scaling in the 'Robots' dataset when varying i and the core count scaling in the 'Example' dataset when varying k, the relevant data is plotted in Figure 5.7. Note that this also means that there may be an inherent dataset specific bias in these plots.



(a) Core count scaling for Robots with $k = 2$.

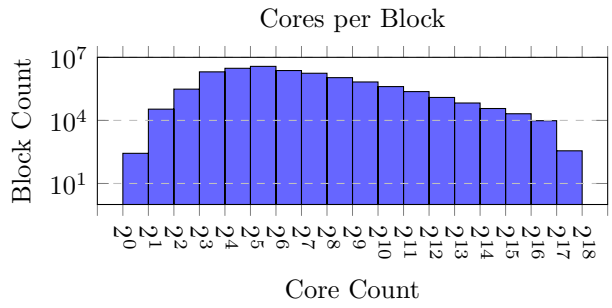
(b) Core count scaling for Example with $i = \infty$.

Figure 5.7: Core count scaling when varying k and i.

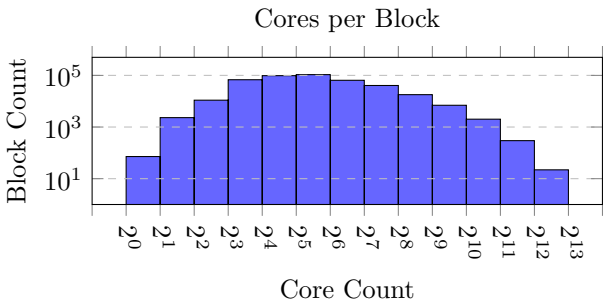
From Figure 5.7a it appears that the core count scales exponentially with the maximum number of intersection. This result is as expected as the algorithm computing intersection cores from Section 4.2.4 is also of an exponential nature. The core count scaling without any intersection limit when varying k in Figure 5.7b appears to be super-exponential. Given the extremely small nature of the 'Example' graph, this means complete indices for large values of k on larger graphs are almost impossible to construct in practise.

5.2.3.2 Core Distribution

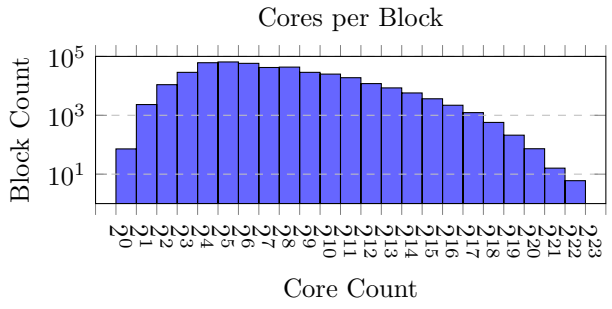
Since we now have a rough idea of the general scaling behind core computation, next we will attempt to get a better idea of where exactly these cores reside in the completed index. Although it would be ideal if all blocks had roughly the same number of cores, this seems extremely unlikely given that some vertices have much more distinctive structure around them than others. For example, in the example graph given in Figure 5.4 in Section 5.2.1.1 the vertex labelled '2' acts as a sort of hub node. All these incoming edges make it possible for many different queries to involve this vertex. The opposite would be single vertices that are only connected to the rest of the graph by a single edge or path of edges, vertices like this have comparatively less identifying structure. Given that many graphs occurring in practice feature these hub nodes we expect that the distribution of cores in a graph is far from even. To get a better idea of what the distribution looks like exactly we have made a histogram for all the datasets with a sufficiently large number of index partitions, meaning the 'Example' dataset is excluded. These histograms are visualised in Figure 5.8. When interpreting these histograms it is important to note the double log scale, which means that the size of the histogram buckets scales exponentially. Note that the log scale also hides buckets with only a single block in them. However, the rightmost bucket always contains at least one block, this is especially relevant for Figures 5.8h, 5.8i and 5.8j.



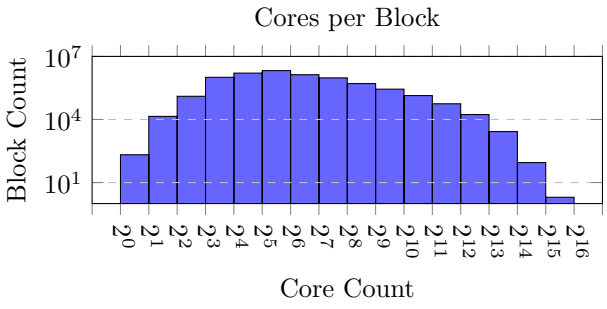
(a) Epinions with $k = 2$ and $i = 2$.



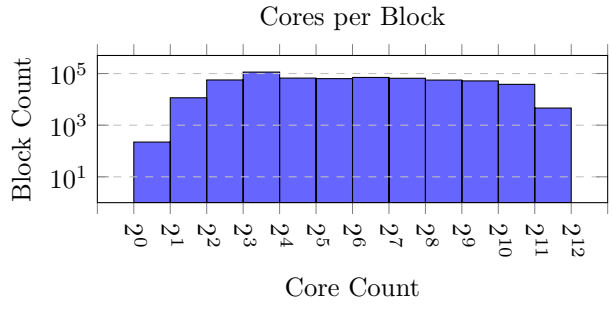
(b) Advogato with $k = 2$ and $i = 2$.



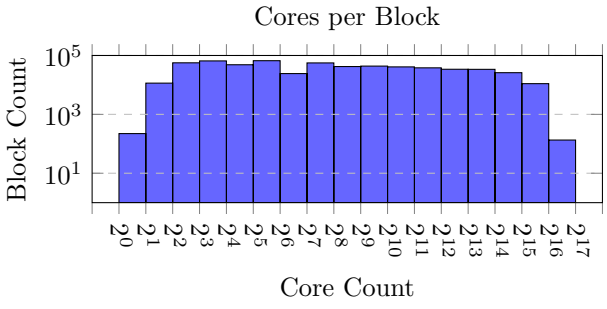
(c) Advogato with $k = 2$ and $i = 4$.



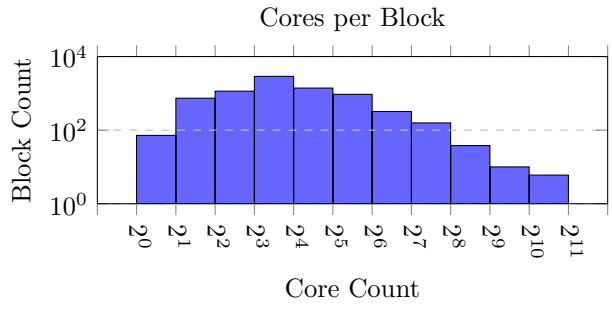
(d) BioGrid with $k = 2$ and $i = 2$.



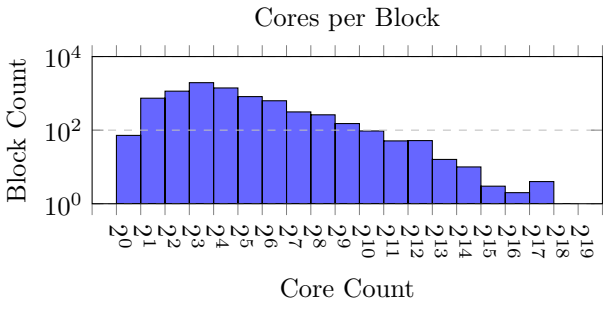
(e) ego-Facebook with $k = 2$ and $i = 2$.



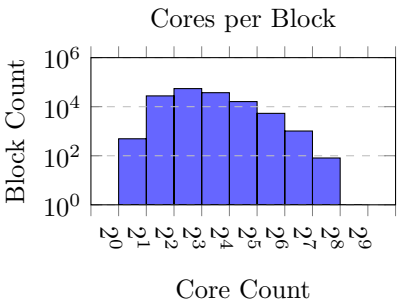
(f) ego-Facebook with $k = 2$ and $i = 3$.



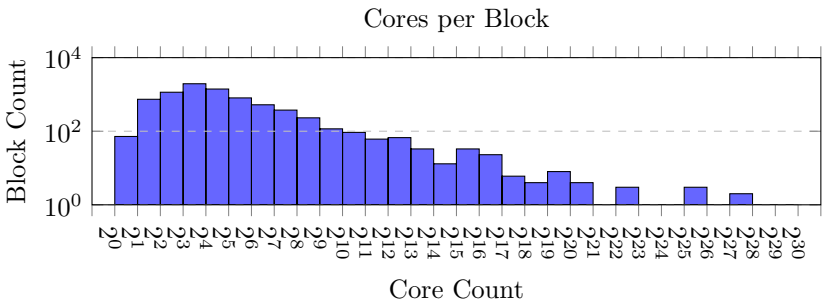
(g) Robots with $k = 2$ and $i = 2$.



(h) Robots with $k = 2$ and $i = 4$.



(i) Robots with $k = 3$ and $i = 1$.



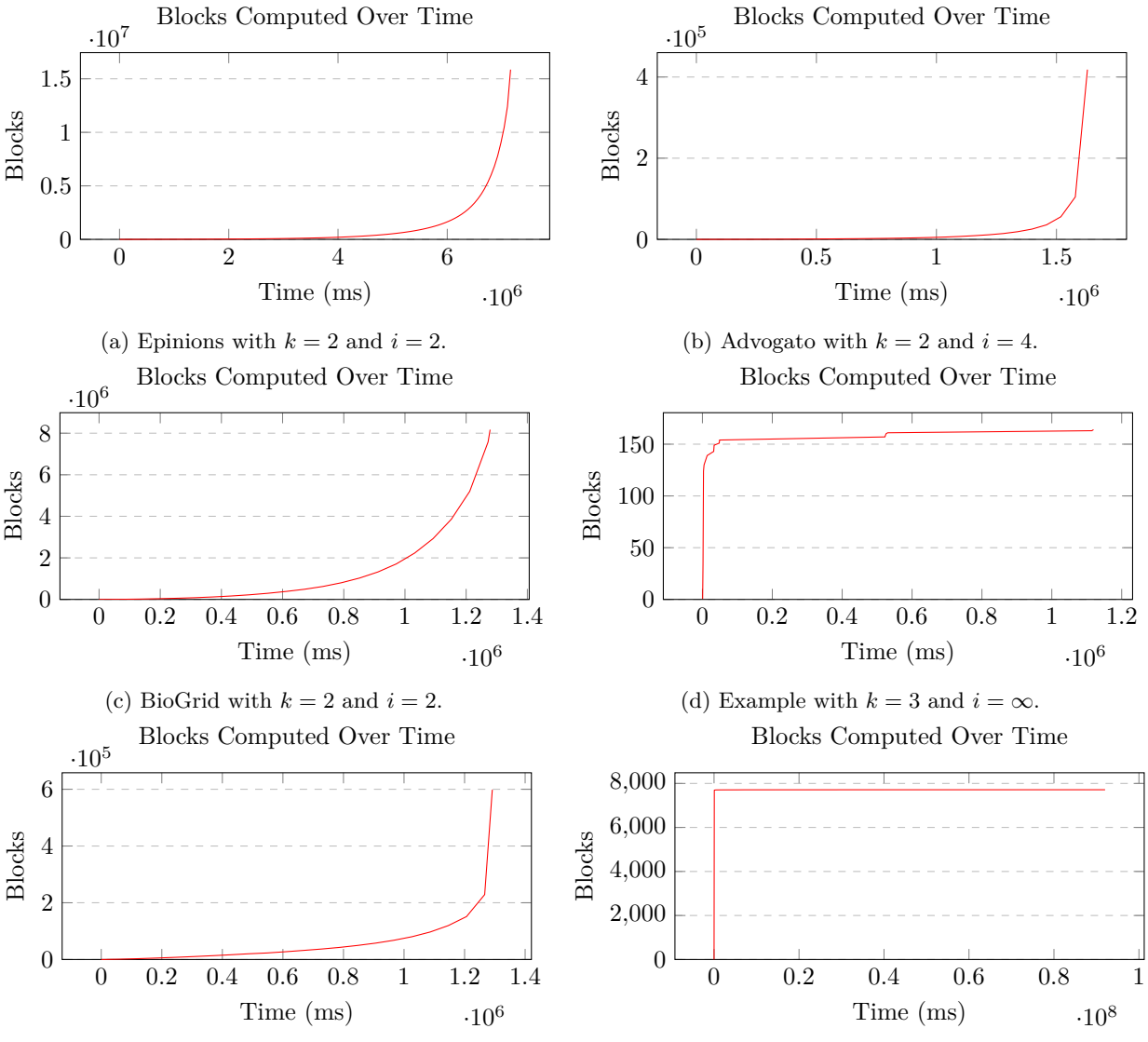
(j) Robots with $k = 2$ and $i = 8$.

Figure 5.8: Core distribution for each dataset.

From these histograms it is clear that the distribution of cores in the index is extremely uneven, with in extreme cases a single block having more cores than all the other blocks in the index combined. However, this result is also not entirely unexpected, since as mentioned, graphs tend to have both extremely well connected nodes and nodes with barely any connections. Unfortunately, this distribution does suggest that our computation strategy of computing blocks in parallel may not be the best fit. Instead it could be beneficial to attempt to find an alternative method where these extremely large blocks can be split up into multiple computation tasks.

5.2.3.3 Block Computation Progress

Since we now know how the cores are distributed through the index, we will next look at how the index computation time is affected by this. Since we need the computation process to take a reasonable amount of time to see any meaningful data, we will focus on runs where core computation took more than 10 minutes. For each of these runs, the total number of blocks that had all their cores computed is plotted against the runtime in Figure 5.9.



(a) Epinions with $k = 2$ and $i = 2$.

(b) Advogato with $k = 2$ and $i = 4$.

(c) BioGrid with $k = 2$ and $i = 2$.

(d) Example with $k = 3$ and $i = \infty$.

(e) ego-Facebook with $k = 2$ and $i = 3$.

(f) Robots with $k = 2$ and $i = 8$.

Figure 5.9: Fully computed blocks over time for the datasets.

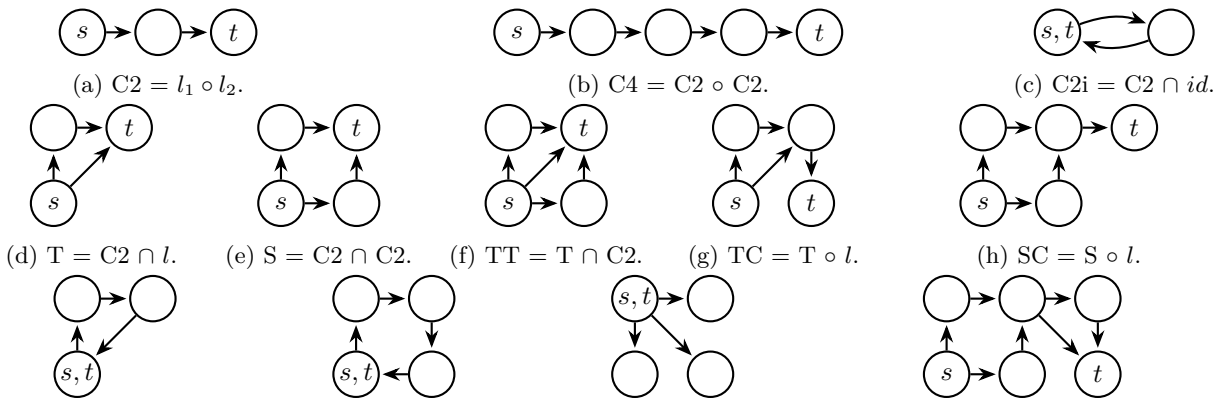
Judging from these runs it appears that often blocks are initially being finishes at a rather slow rate, followed by a gradual increase in speed towards the end. This makes sense given that we know that some blocks have significantly more cores than other blocks from Section 5.2.3.2. Moreover, this result is expected as it was observed during testing that large blocks are grouped together during sorting by the partitioning step. We take advantage of this by starting the processing on the largest blocks to improve the work distribution. This means that the blocks get progressively less expensive to compute. However, there are also more extreme cases. For example, for 'Robots' in Figure 5.9f all blocks except for one finish computing within about 1 hour, after which

the final block takes the remaining 24 hours of the runtime. Based on the histogram from Figure 5.8j in Section 5.2.3.2 this makes sense, as this block has between 2^{29} and 2^{30} cores. However, given that 63 CPU threads were idle while this block was being computed, a better work distribution could be beneficial.

5.3 Query Evaluation

Finally, we will end with some notes on query evaluation performance, for which we did not run any tests. Note that query performance for the current index is determined entirely by a simple hash map lookup after the core of the input query is computed. An expectation for the core computation process time was already established in Section 5.1 and the performance of the hash map implementation in Java is not very interesting to discuss here. If the index gets extended to handle queries not strictly within its parameters, something which we will briefly discuss in Section 6.1.3, then query evaluation will become an important component to run tests for.

That being said, given that we have a rough idea of the practical limitations of the index from the results in Section 5.2, we can look at the applicability of the index to common query workloads. In the paper by Sasaki et al. [65] the performance of index querying is evaluated using twelve *CPQ* query templates representative of structures that frequently appear in real world use cases [65, 14, 13]. These query templates are given in Figure 5.10 and are representative of chains, stars, cycles and flowers. Note that edge labels have been omitted from these templates, but a concrete initialisation of one of these templates would have concrete edge labels on all of its edges.



(a) C2 = $l_1 \circ l_2$. (b) C4 = C2 $\circ$ C2. (c) C2i = C2 $\cap$ *id*.

(d) T = C2 $\cap$ *l*. (e) S = C2 $\cap$ C2. (f) TT = T $\cap$ C2. (g) TC = T $\circ$ *l*. (h) SC = S $\circ$ *l*.

(i) Ti = (C2 $\circ$ *l*) $\cap$ *id*. (j) Si = C4 $\cap$ *id*. (k) St = $(l_1 \circ l_1^-) \cap (l_2 \circ l_2^-) \cap (l_3 \circ l_3^-) \cap id$). (l) ST = S $\circ$ T

Figure 5.10: *CPQ* query templates, only source and target are indicated and edge labels are left free.

Before we can determine which of these templates can readily be accelerated using the current index design, we first need to determine the diameter k and maximum number of intersections for each of these query templates. For all of the templates these properties have been compiled in Table 5.6. Recall that intersection with identity does not count towards the maximum intersections limit, no intersections is indicated in the table as 1.

Table 5.6: Overview of query template attributes.

Template	**Figure**	**Diameter**	**Intersections**
C2	Figure 5.10a	2	1
C4	Figure 5.10b	4	1
C2i	Figure 5.10c	2	1
T	Figure 5.10d	2	2
S	Figure 5.10e	2	2
TT	Figure 5.10f	2	3
TC	Figure 5.10g	3	2
SC	Figure 5.10h	3	2
Ti	Figure 5.10i	3	1
Si	Figure 5.10j	4	1
St	Figure 5.10l	2	3
ST	Figure 5.10k	4	2

Based on the data in this table, we can conclude the following. First of all, note that six of the query templates have a diameter k that is greater than 2. Even ignoring the fact that computing cores for $k > 2$ has proved challenging, computing the base partitions required for these query templates is already a challenge. Thus it is likely that these queries will instead have to be decomposed into smaller diameter parts before the index can be used to accelerate them. Secondly, note that a majority of the templates has a low maximum intersection count, with only two queries requiring a limit of 3. This is a promising result as computing cores for an intersection limit of 2 generally seems feasible without significant overhead. Overall, this means that even with low limits like $k = 2$ and $i = 3$ already half of these templates can be retrieved directly from a constructed index. Note that the remaining query templates can be evaluated using exactly two index retrievals.

5.4 Discussion

Within this chapter we have presented an exhaustive performance evaluation of the proposed index on a number of datasets representative of real world use cases. Although the proposed index has practical limitations, the index is also promising in more restrictive configurations. In this section we will briefly summarise the results of the performance evaluation and compile all the strengths and weaknesses of the *CPQ*-native Index.

As mentioned in Section 5.2.2, partitioning currently often takes a significant amount of time on larger datasets. However, as mentioned this step was not a focus for optimisation in this thesis, so significant performance improvements are likely possible. Nevertheless, it is worth noting that as shown in Section 5.2.3 core computation for low values of k and i is often significantly faster than the partitioning step. This means that computing at least some cores does not need to impose significant construction overhead. Moreover, under these limits core computation appears to follow a surprisingly linear trend in the number of vertices and edges in the dataset as shown in Section 5.2.3.1.

Unfortunately, in Section 5.2.3.1 we also find that scaling in the limiting parameters k and i themselves is not very promising. The exponential scaling in the number of cores when increasing i presents a serious scalability concern and the super-exponential scaling in the number of cores when increasing k makes indices for a large value of k practically impossible to compute. In fact, these trends mean that available RAM memory is more likely to become an issue instead of the increase in runtime. Exacerbating this is issue the incredibly uneven distribution of cores throughout the index as shown in Section 5.2.3.2. It is far from ideal that often a few large blocks we may never even need for query evaluation are contributing the majority of the cores in the entire index. Furthermore, these blocks are also the root cause of the suboptimal block computation progression shown in Section 5.2.3.3.

Nevertheless, as highlighted in Section 5.3, while the aforementioned issues are worthy of consideration, they do not present an immediate usability concern. Many of the *CPQ*s appearing in practice are small enough that they can be accelerated using an index operating under extremely restrictive limits. The number of intersections in particular generally appears to be below 3. Instead, the larger diameter required for some queries is a more serious cause of concern given the aforementioned scaling issue related to k. However, as noted, the query templates that were presented can all be split such that they can be accelerated by the index in just two lookups. We therefore conclude that a limited form of native support for *CPQ*s can realistically be provided by a graph database index and that this is a valuable asset in real world use cases.

Concluding Remarks

6 第六章

In this thesis we have set out to create a *CPQ*-native graph database index. During this process we have both achieved our original goal of creating this index, as well as created several components that are valuable contributions in their own right. In Chapter 2 we started by introducing basic concepts and terminology required to build the index. Then in Chapter 3 we covered computing the core of a *CPQ*, the most important building block for the proposed index. Moreover, in the process we developed new algorithms for computing tree decompositions and for computing graph cores, we also adapted an existing query containment algorithm and designed a strategy for computing canonical forms with nauty. Next in Chapter 4 we then finally introduced the *CPQ*-native Index and extensively documented how the cores for each block can be computed, we also presented a slight modification to the original partitioning method to support *CPQ* core computation. Finally, in Chapter 5 we presented an extensive evaluation of the performance of the created index.

Overall, we have not only succeeded in creating a *CPQ*-native Index, but we have also shown in Section 5.3 that it can realistically be deployed to accelerate common queries as long as proper limits are configured. Thereby successfully answering our problem statement of how to index *CPQ*s to accelerate query evaluation, and accomplishing our goal of creating an index that can be deployed for real world use cases. Nevertheless, there are still clear limitations to the approach presented in this thesis. Large diameters for the index result in serious computation issues and the same holds true for intersections if not limited. As soon as a single block in a layer builds up a sizeable number of cores, it is unlikely that the next layer can still be computed due to the exponential and quadratic nature of several involved algorithms. Ultimately, there is still a lot of work that can be done to either improve or augment the proposed index and we will discuss some of these future research directions in Section 6.1. However, the presented index can also be used as inspiration to develop indices for other query languages. As mentioned in Section 2.3, both the index[1] and all the individual utilities[2] are open-source and available on GitHub [37, 36] for further research and development.

6.1 Future Work

As already mentioned in various places throughout this thesis, there are many components that could still be improved. The main goal of this thesis was to explore the feasibility and practical limits of an index completely based around k-path-bisimulation and *CPQ* cores, without sacrificing coverage or query performance. However, as we have seen in Chapter 5, this does mean that the current index has a number of practical limitations. In this section we will present some directions for future work, some of these items focus on improving the performance of the current approach, while others are alternative directions based on certain trade-offs.

6.1.1 Query Homomorphism Testing

The current implementation of the query homomorphism testing approach presented in Section 3.1.3 admits an important optimisation. Currently the approach makes use of the fallback method from the paper where a tree decomposition is computed from the query incidence graph instead of a query decomposition [17]. This was done because currently no algorithms are known that can directly compute a query decomposition. However, the authors of this paper suspect that computing a query decomposition is easier than computing a tree decomposition. Moreover, *CPQ*s are of treewidth 2, which may mean that a query decomposition can efficiently be computed for them. Developing an algorithm to compute *CPQ* query decompositions could be done to achieve better query containment testing times. This is especially significant due to the large number of query containment checks performed for core computation during index construction. However, it is worth noting that tree decomposition and incidence graph computation are not currently the most expensive components of the core computation algorithm.

[1] `https://github.com/RoanH/CPQ-native-index`
[2] `https://github.com/RoanH/gMark`

6.1.2 Custom Canonisation

Currently nauty [54] is used to compute canonical labellings of *CPQ* query graphs. However, as shown in Section 3.4.2, this requires several transformations of the input graph. Moreover, nauty has been developed for the general case, meaning its runtime may not be best for *CPQ*s which are of treewidth 2 and thus likely admit a tailored more optimised approach. Therefore, there may be merit in attempting to develop a canonisation algorithm specifically tailored towards *CPQ*s to further improve the performance of canonisation. More recommendations around this topic can be found in the *CPQ* Keys report [35].

6.1.3 Beyond *CPQ*s

Currently the index can only be used to accelerate the processing of queries of at most a specific diameter k. However, in practice we will likely often need to evaluate queries that have a larger diameter than the diameter the index was constructed for, or even queries that are not *CPQ*s at all. Moreover, even for queries like this we would like to leverage the index we have already constructed as much as possible. In order to process queries like this we need to decompose them into a set of index compatible CPQ_k queries that can be evaluated individually and then have their results combined. For example, if we have a long path we can split it into smaller paths of a given maximum length and then simply chain the results. The language-aware path index [65] does something similar, where any input query is covered by paths. For the *CPQ*-native Index we want to cover our input *CPQ*s with CPQ_k queries instead. Also note that the intersection limit of the index should be respected during this process. Researching the most efficient way to cover general queries like this is an important direction for future work.

6.1.4 Better Multi-threading

As noted in Section 5.2.3.3 the runtime of core computation during index construction can sometimes be dominated by a single block. Ideally we would want to make it possible for multiple CPU threads to work on the same block at the same time to prevent this from happening. For example, for the example with the 'Robots' dataset with $k = 2$ and $i = 8$, the last block took 1 thread 24 hours with 63 other threads being idle. This means that if all threads could work on this block at the same time, the runtime for index construction could be reduced by 23.5 hours. In this particular example that would mean reducing the total runtime of index core computation by 83%.

6.1.5 Core Computation Outliers

As noted in Section 5.1.2 major outliers are an important issue for the current core computation algorithm. It would be worthwhile to investigate if there is any prevalent structure in these outliers that we can take advantage of to achieve a better average runtime with less variance.

6.1.6 Accepting Superfluous Cores

By far the most time during block computation is spent computing the core of candidate *CPQ*s. However, note that many candidate *CPQ*s are either already cores or their core will also be computed via an alternative route. This makes it theoretically possible to simply accept these non cores and store them as if they were cores, meaning we can completely skip most core computations. It is an interesting research question if this trade-off is worth it as potentially much more storage may be required for some datasets.

6.1.7 Optimise Partitioning

As clearly visible in Section 5.2.2, the partitioning performance of the current reference implementation is relatively slow. Recall that there were optimisations in the original language-aware index [65] that we did not implement. However, some of these optimisations should still be applicable and could therefore result in significantly better partitioning performance.

6.1.8 Input Query Splitting

As clearly visible in Section 4.2.4, intersection cores contribute the most queries to the complete index. As such, one method to mitigate this issue is to split input queries into the individual main paths that would otherwise be stored as intersections for the top layer of the index. We then simply look up each of these individual queries and combine the results. Note though that this method negatively affects query performance. However, this trade-off may be worth it for certain applications.

6.1.9 Optimise Core Computation

As already mentioned at some points during Section 4.2, there are currently still a number of inefficiencies in the general method that lead to more cores being computed than strictly required. Obtaining a tighter bound on the algorithms used to compute candidate cores can greatly reduce the time required to compute cores for the index. In particular, if we can prove a bound to be tight to the point that no superfluous cores are computed at all, then we can skip the expensive core computation algorithm and only perform core canonisation. Currently, as already mentioned one of the core problems is the issue of backflow mentioned in Section 4.2.4. Another issue already mentioned is the problem surrounding identity in join cores brought up in Section 4.2.3. This issue actually also gives rise to a second problem. Note that the join of two cores where the source and target are the same node, is essentially the same as taking the intersection of the cores. This means that for blocks representing looping paths, some of the join cores are intersections already, leading to more work for the intersection algorithm. However, note that fortunately this problem is limited to blocks representing loops.

6.1.10 Index Maintenance

Within this thesis we did not thoroughly investigate if the *CPQ*-naive Index can be maintained efficiently. However, we do know from the extended report [64] for the language-aware indexing paper that the language-aware index can be maintained efficiently. Note that cores are computed per block, so in theory we can just recompute the cores for only the blocks that changed during an update. However, it remains to be seen if the update method for the partitioning is compatible with the information required for core computation.

6.1.11 Interest Awareness

As noted in Section 4.5, limiting intersections is already a limited form of interest awareness where we give the user some degree of control over which cores they want the index to store. However, giving the user full control over which cores they want in the index is a major step towards a more scalable index. More information about this can also be found in the language-aware indexing paper [65] as an interest-aware variant of the path index is also proposed.

6.1.12 Topology Awareness

As discussed in Section 5.2.3.2 and 5.2.3.3, vertices with many connections tend to contribute the largest number of cores in the index. Consequently, this also means that these vertices dominate the runtime for core computation. This situation is not ideal as we do not want to be dedicating a majority of the memory and computation resources to single blocks that are generally few in number. Instead, this suggests that the index should be aware of the topology of the graph. Once the index is topology aware we can then apply specialised data structures or limits on a per block basis. For example, we can consider data structures that do not explicitly materialise the entire core set for a block and instead do extra work at query time, or core construction limits that only apply to specific blocks. This extension would make it easier to adapt the index to specific database graph and complements interest awareness discussed in Section 6.1.11.

Bibliography

[1] Luca Aceto, Anna Ingolfsdottir, and Jirí Srba. "The algorithmics of bisimilarity". In: *Advanced Topics in Bisimulation and Coinduction*. Ed. by Davide Sangiorgi and Jan Rutten. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2011, pp. 100–172. DOI: 10.1017/CBO9780511792588.004.

[2] 秋葉 拓哉 (Takuya Akiba) and 岩田 陽一 (Yoichi Iwata). "姉妹港 (Sister Ports). ACM-ICPC OB/OG の会 2012 年模擬国内予選 問題 G (ACM-ICPC OB/OG 2012 Domestic Qualifiers Practise Problem G)". Japanese. ICPC OB/OG, June 17, 2012. URL: https://jag-icpc.org/?plugin=attach&refer=2012%2FPractice%2F%E6%A8%A1%E6%93%AC%E5%9B%BD%E5%86%85%E4%BA%88%E9%81%B8%2F%E8%AC%9B%E8%A9%95&openfile=ports.pdf (visited on 2023-03-02).

[3] "Apache License". Version 2.0. Apache Software Foundation, Jan. 2004. URL: https://www.apache.org/licenses/LICENSE-2.0.

[4] László Babai. "Graph Isomorphism in Quasipolynomial Time". In: *CoRR* abs/1512.03547 (2015). DOI: 10.48550/ARXIV.1512.03547. arXiv: 1512.03547.

[5] László Babai, William M Kantor, and Eugene M Luks. "Computational complexity and the classification of finite simple groups". In: *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. 1983, pp. 162–171. DOI: 10.1109/SFCS.1983.10.

[6] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. "gMark: Schema-Driven Generation of Graphs and Queries". In: *IEEE Transactions on Knowledge and Data Engineering* 29.4 (2017), pp. 856–869.

[7] Umberto Bertelè and Francesco Brioschi. "Nonserial Dynamic Programming". In: *Mathematics in Science and Engineering* 91 (1972). URL: https://www.sciencedirect.com/bookseries/mathematics-in-science-and-engineering/vol/91.

[8] BioGRID. "BioGRID | Database of Protein, Genetic and Chemical Interactions". URL: https://thebiogrid.org/.

[9] Hans L. Bodlaender. "A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth". In: *SIAM Journal on Computing* 25.6 (1996), pp. 1305–1317. DOI: 10.1137/S0097539793251219.

[10] Hans L. Bodlaender. "A partial k-arboretum of graphs with bounded treewidth". In: *Theoretical Computer Science* 209.1 (1998), pp. 1–45. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(97)00228-4.

[11] Hans L. Bodlaender and Ton Kloks. "Efficient and Constructive Algorithms for the Pathwidth and Treewidth of Graphs". In: *Journal of Algorithms* 21.2 (1996), pp. 358–402. ISSN: 0196-6774. DOI: https://doi.org/10.1006/jagm.1996.0049.

[12] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. "Querying graphs". In: *Synthesis Lectures on Data Management* 10.3 (2018), pp. 1–184.

[13] Angela Bonifati, Wim Martens, and Thomas Timm. "An analytical study of large SPARQL query logs". In: *The VLDB Journal* 29.2 (May 1, 2020), pp. 655–679. ISSN: 0949-877X. DOI: 10.1007/s00778-019-00558-9. URL: https://doi.org/10.1007/s00778-019-00558-9.

[14] Angela Bonifati, Wim Martens, and Thomas Timm. "Navigating the Maze of Wikidata Query Logs". In: *The World Wide Web Conference*. WWW '19. San Francisco, CA, USA: Association for Computing Machinery, 2019, pp. 127–138. ISBN: 9781450366748. DOI: 10.1145/3308558.3313472. URL: https://doi.org/10.1145/3308558.3313472.

[15] Richard B. Borie, R. Gary Parker, and Craig A. Tovey. "Solving Problems on Recursively Constructed Graphs". In: *ACM Comput. Surv.* 41.1 (Jan. 2009). ISSN: 0360-0300. DOI: 10.1145/1456650.1456654. URL: https://doi.org/10.1145/1456650.1456654.

[16] Wieb Bosma, John Cannon, and Catherine Playoust. "The Magma algebra system. I. The user language". In: *J. Symbolic Comput.* 24.3-4 (1997). Computational algebra and number theory (London, 1993), pp. 235–265. ISSN: 0747-7171. DOI: 10.1006/jsco.1996.0125.

[17] Chandra Chekuri and Anand Rajaraman. "Conjunctive query containment revisited". In: *Theoretical Computer Science* 239.2 (2000), pp. 211–229. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(99)00220-0.

[18] Derek G. Corneil and Calvin C. Gotlieb. "An Efficient Algorithm for Graph Isomorphism". In: *Journal of the ACM (JACM)* 17.1 (Jan. 1970), pp. 51–64. ISSN: 0004-5411. DOI: 10.1145/321556.321562.

[19] Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. "Exploiting Structure in Symmetry Detection for CNF". In: *Proceedings of the 41st Annual Design Automation Conference*. DAC '04. San Diego, CA, USA: Association for Computing Machinery, 2004, pp. 530–534. ISBN: 1581138288. DOI: 10.1145/996566.996712.

[20] Daniel J. Dougherty and Claudio Gutiérrez. "Normal forms for binary relations". In: *Theoretical Computer Science* 360.1 (2006), pp. 228–246. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2006.03.023. URL: https://www.sciencedirect.com/science/article/pii/S0304397506002787.

[21] Eclipse Foundation. *Eclipse Temurin*. Version 17.0.7+7. Apr. 18, 2023. URL: https://adoptium.net/.

[22] ei1333. "木幅が 2 以下のグラフの木分解と動的計画法 (Tree Decompositions and Dynamic Programming for Graphs with Treewidth 2 or less)". Japanese. Feb. 12, 2020. URL: https://ei1333.hateblo.jp/entry/2020/02/12/150319 (visited on 2023-03-02).

[23] George Fletcher. "Notes on conjunctive path queries". Internal Notes. Apr. 6, 2020.

[24] George H. L. Fletcher, Marc Gyssens, Dirk Leinders, Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. "Similarity and bisimilarity notions appropriate for characterizing indistinguishability in fragments of the calculus of relations". In: *Journal of Logic and Computation* 25.3 (Mar. 2014), pp. 549–580. ISSN: 0955-792X. DOI: 10.1093/logcom/exu018. eprint: https://academic.oup.com/logcom/article-pdf/25/3/549/6138494/exu018.pdf. URL: https://doi.org/10.1093/logcom/exu018.

[25] George H. L. Fletcher, Marc Gyssens, Dirk Leinders, Jan Van den Bussche, Dirk Van Gucht, Stijn Vansummeren, and Yuqing Wu. "Relative Expressive Power of Navigational Querying on Graphs". In: *Proceedings of the 14th International Conference on Database Theory*. ICDT '11. Uppsala, Sweden: Association for Computing Machinery, 2011, pp. 197–207. ISBN: 9781450305297. DOI: 10.1145/1938551.1938578. URL: https://doi.org/10.1145/1938551.1938578.

[26] George H. L. Fletcher and Stijn Vansummeren. "Homomorphisms on series-parallel graphs with inverse and identity". June 16, 2014. preprint.

[27] *GAP – Groups, Algorithms, and Programming*. The GAP Group. URL: https://www.gap-system.org.

[28] "GNU General Public License". Version 3. Free Software Foundation, June 29, 2007. URL: http://www.gnu.org/licenses/gpl.html.

[29] Oded Goldreich, Silvio Micali, and Avi Wigderson. "Proofs That Yield Nothing but Their Validity or All Languages in NP Have Zero-Knowledge Proof Systems". In: *J. ACM* 38.3 (July 1991), pp. 690–728. ISSN: 0004-5411. DOI: 10.1145/116825.116852.

[30] Martin Grohe, Daniel Neuen, Pascal Schweitzer, and Daniel Wiebking. "An Improved Isomorphism Test for Bounded-Tree-Width Graphs". In: *ACM Trans. Algorithms* 16.3 (June 2020). ISSN: 1549-6325. DOI: 10.1145/3382082.

[31] Rudolf Halin. "S-functions for graphs". In: *Journal of Geometry* 8.1 (Mar. 1976), pp. 171–186. DOI: 10.1007/BF01917434.

[32] Pavol Hell and Jaroslav Nešetřil. "The core of a graph". In: *Discrete Mathematics* 109.1 (1992), pp. 117–126. ISSN: 0012-365X. DOI: https://doi.org/10.1016/0012-365X(92)90282-K.

[33] 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.

[34] Roan Hofland. "CPQ Keys | Notes on CPQ key generation and canonization". July 18, 2022. URL: https://cpqkeys.roanh.dev/.

[35] 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.

[36] Roan Hofland. *CPQ-native Index*. Version 1.0. July 12, 2023. URL: https://github.com/RoanH/CPQ-native-index.

[37] Roan Hofland. *gMark*. Version 1.2. July 5, 2023. URL: https://github.com/RoanH/gMark.

[38] Roan Hofland. *gMark*. Version ee76a39ec308d9cfec295ecbe307368dea2a18a6. May 28, 2023. URL: https://github.com/RoanH/gMark.

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

[40] Roan Hofland. "Treewidth | CPQ Keys". July 18, 2022. URL: https://cpqkeys.roanh.dev/notes/treewidth.

[41] Thor Johnson, Neil Robertson, P.D. Seymour, and Robin Thomas. "Directed Tree-Width". In: *Journal of Combinatorial Theory, Series B* 82.1 (2001), pp. 138–154. ISSN: 0095-8956. DOI: 10.1006/jctb.2000.2031.

[42] Simon Josefsson. "The Base16, Base32, and Base64 Data Encodings". RFC 4648. Oct. 2006. DOI: 10.17487/RFC4648. URL: https://www.rfc-editor.org/info/rfc4648.

[43] Paris C. Kanellakis and Scott A. Smolka. "CCS expressions, finite state processes, and three problems of equivalence". In: *Information and Computation* 86.1 (1990), pp. 43–68. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(90)90025-D. URL: https://www.sciencedirect.com/science/article/pii/089054019090025D.

[44] "Pathwidth of pathwidth-bounded graphs". In: *Treewidth: Computations and Approximations*. Ed. by Ton Kloks. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 147–172. ISBN: 978-3-540-48672-5. DOI: 10.1007/BFb0045388. URL: https://doi.org/10.1007/BFb0045388.

[45] Jérôme Kunegis. "KONECT: the Koblenz network collection". In: *Proceedings of the 22nd International Conference on World Wide Web* (2013).

[46] Seiji Maekawa, Yuya Sasaki, George H. L. Fletcher, and Makoto Onizuka. "Calculating the Number of Cores in CPQ". Internal Notes. Feb. 21, 2022.

[47] Seiji Maekawa, Yuya Sasaki, George H. L. Fletcher, and Makoto Onizuka. "CPQ-Core-based Index Design". Internal Notes. Sept. 9, 2022.

[48] Brendan D. McKay. "Backtrack Programming and the Graph Isomorphism Problem". MA thesis. University of Melbourne, July 1976. URL: https://users.cecs.anu.edu.au/~bdm/papers/McKayMastersThesis.pdf.

[49] Brendan D. McKay. "Computing automorphisms and canonical labellings of graphs". In: *Combinatorial Mathematics*. Ed. by D. A. Holton and Jennifer Seberry. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 223–232. ISBN: 978-3-540-35702-5. DOI: 10.1007/BFb0062536.

[50] Brendan D. McKay. "Labeled graphs". Message to the Nauty mailing list. Sept. 8, 2013. URL: https://mailman.anu.edu.au/pipermail/nauty/2013-September/000693.html.

[51] Brendan D. McKay. "Practical Graph Isomorphism". In: *Congressus Numerantium* 30 (1981), pp. 45–87. URL: http://users.cecs.anu.edu.au/~bdm/papers/pgi.pdf.

[52] Brendan D. McKay and Adolfo Piperno. *nauty & Traces*. Version 2.8.6. Nov. 2022. URL: https://pallini.di.uniroma1.it/.

[53] Brendan D. McKay and Adolfo Piperno. *nauty and Traces User's Guide (Version 2.8.6)*. Nov. 16, 2022. URL: https://pallini.di.uniroma1.it/nug28.pdf.

[54] Brendan D. McKay and Adolfo Piperno. "Practical graph isomorphism, II". In: *Journal of Symbolic Computation* 60 (2014), pp. 94–112. ISSN: 0747-7171. DOI: https://doi.org/10.1016/j.jsc.2013.09.003.

[55] "MIT License". URL: https://spdx.org/licenses/MIT.html.

[56] Oracle. *Java Native Interface Specification*. URL: https://docs.oracle.com/en/java/javase/18/docs/specs/jni/index.html.

[57] Robert Paige and Robert E. Tarjan. "Three Partition Refinement Algorithms". In: *SIAM Journal on Computing* 16.6 (1987), pp. 973–989. DOI: 10.1137/0216062. eprint: https://doi.org/10.1137/0216062. URL: https://doi.org/10.1137/0216062.

[58] R Parris and Ronald C. Read. "A coding procedure for graphs". In: *Scientific Report. UWI/CC* 10 (1969).

[59] You Peng, Ying Zhang, Xuemin Lin, Lu Qin, and Wenjie Zhang. "Answering Billion Scale Label Constrained Reachability Queries within Microsecond". In: *Proc. VLDB Endow.* 13.6 (Feb. 2020), pp. 812–825. ISSN: 2150-8097. DOI: 10.14778/3380750.3380753. URL: https://doi.org/10.14778/3380750.3380753.

[60] Adolfo Piperno. "Search Space Contraction in Canonical Labeling of Graphs". In: *CoRR* abs/0804.4881 (2008). DOI: 10.48550/ARXIV.0804.4881. arXiv: 0804.4881.

[61] Neil Robertson and P.D Seymour. "Graph minors. II. Algorithmic aspects of tree-width". In: *Journal of Algorithms* 7.3 (1986), pp. 309–322. ISSN: 0196-6774. DOI: 10.1016/0196-6774(86)90023-4.

[62] The Sage Developers. *SageMath, the Sage Mathematics Software System*. URL: https://www.sagemath.org.

[63] Yuya Sasaki. *CPQ-aware Index for Conjunctive Path Queries*. Version 67409f2. Mar. 12, 2021. URL: https://github.com/yuya-s/CPQ-aware-index.

[64] Yuya Sasaki, George Fletcher, and Makoto Onizuka. "Extended report. Language-aware Indexing for Conjunctive Path Queries". In: (Nov. 4, 2021). URL: https://github.com/yuya-s/CPQ-aware-index/blob/main/extended_version_sCPQ-aware_indexing.pdf.

[65] 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.

[66] SNAP. "Stanford Network Analysis Project". Stanford University. URL: https://snap.stanford.edu/.

[67] trustlet.org. "Robots". July 7, 2014. URL: http://www.trustlet.org/datasets/robots_net/robots_net-graph-2014-07-07.dot.

[68] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. "The Recognition of Series Parallel Digraphs". In: *SIAM Journal on Computing* 11.2 (1982), pp. 298–313. DOI: 10.1137/0211023.

[69] Lucien D.J. Valstar, George H.L. Fletcher, and Yuichi Yoshida. "Landmark Indexing for Evaluation of Label-Constrained Reachability Queries". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 345–358. ISBN: 9781450341974. DOI: 10.1145/3035918.3035955. URL: https://doi.org/10.1145/3035918.3035955.

[70] Arlazarov Vladimir Lvovich, I.I. Zuev, A.V. Uskov, and I.A. Faradzhev. "An algorithm for the reduction of finite non-oriented graphs to canonical form". In: *USSR Computational Mathematics and Mathematical Physics* 14.3 (1974), pp. 195–201. ISSN: 0041-5553. DOI: 10.1016/0041-5553(74)90114-1.

Plain *CPQ* Generation

A

This appendix contains the random *CPQ* generation algorithm originally presented in Algorithm 1 in Section 2.5.1 of the CPQ Keys report [35]. The algorithm is relatively straightforward and shown in Algorithm 6. The original motivation behind this algorithm was to provide a significantly more efficient and random method of generating random *CPQ*s than the schema driven method that was originally developed for gMark [33]. Note that this original approach was limited in its randomness due to its adherence to a specific graph schema, while the new algorithm only follows the rules of the *CPQ* grammar from Section 2.2.2.

Algorithm 6 Conjunctive Path Query Generation

```
1: procedure GeneratePlainCPQ(r ∈ ℕ, i ∈ {true, false}, ℒ) ▷ Rule applications, allow identity, labels
2:     if r = 0 then
3:         if i ∧ RandomBoolean() = true then
4:             return id
5:         else
6:             l ← SelectRandom(ℒ)
7:             if RandomBoolean() = true then
8:                 return l
9:             else
10:                return l⁻
11:            end if
12:        end if
13:    else
14:        s ← UniformRandom(0, r − 1)
15:        if RandomBoolean() = true then
16:            return GeneratePlainCPQ(s, false, ℒ) ∘ GeneratePlainCPQ(r − s − 1, false, ℒ)
17:        else
18:            return GeneratePlainCPQ(s, true, ℒ) ∩ GeneratePlainCPQ(r − s − 1, false, ℒ)
19:        end if
20:    end if
21: end procedure
```

It should be noted that this algorithm intentionally prevents two technically valid but otherwise uninteresting *CPQ* patterns from being generated. The first is the conjunction of the identity operation with itself, given that $id \cap id = id$. The second is the join of identity with another *CPQ*, for a *CPQ* q this would result in $q \circ id = q$. The algorithm takes three arguments as input. The first r is the number of join and conjunction rule applications. From the *CPQ* grammar, this refers to the exact number of times the grammar steps containing join '$\circ$' and conjunction '$\cap$' are applied. The second argument i is a boolean indicating whether it is allowed for the algorithm to return the *CPQ* containing only the identity operation. The last argument $\mathcal{L}$ is a set of labels that the algorithm is allowed to pick from to form the *CPQ*. This set of labels should not include inverse labels.

For the initial call to the algorithm the value of i should be set to **false**. Not doing this makes it possible for the *CPQ* consisting of only *id* to be generated, which for most applications will be an undesired query. The set of labels $\mathcal{L}$ should also contain at least one label. There are no restrictions on r as long as it is a natural number (zero is allowed).

gMark Setup

B

This appendix will detail how to use and obtain a copy of the updated gMark software. The software is released on GitHub[1] and the README file of the repository will always contain the most up-to-date instructions on using the software. This appendix is written for release 1.2 of gMark [37] and may not be fully correct for newer releases of gMark. To support a wide variety of use cases, gMark is a available in a number of different formats, including as a docker image[2] and as a standalone executable. However, these release types are only relevant if the intent is to use gMark to generate query workloads. To make use of all the *CPQ* related features, gMark should either be used as a maven dependency or be developed directly.

B.1 Usage Example

Most of the *CPQ* related utilities are accessible via the `CPQ` class and all general utilities are located in the `dev.roanh.gmark.util` package. For example, the following snippet in Listing B.1 shows two methods to construct the *CPQ* $a \cap a$, how to generate a random *CPQ*, and how to compute the core of each.

Listing B.1: Usage example for the *CPQ* API in gMark.

```
Predicate a = new Predicate(0, "a");
QueryGraphCPQ core;

core = CPQ.parse("a ∩ a").computeCore();
core = CPQ.intersect(a, a).toQueryGraph().computeCore();
core = CPQ.generateRandomCPQ(4, 1).toQueryGraph().computeCore();
```

B.2 Maven Artifact

An artifact[3] for gMark is available on maven central, so it can be included directly in another Java project using build tools like Gradle and Maven. This way it becomes possible to directly use all the implemented constructs and utilities. A hosted version of the javadoc for gMark can be found at gmark.docs.roanh.dev[4].

Listing B.2: Gradle setup for gMark.

```
repositories{
    mavenCentral()
}

dependencies{
    implementation 'dev.roanh.gmark:gmark:1.2'
}
```

Listing B.3: Maven setup for gMark.

```
<dependency>
    <groupId>dev.roanh.gmark</groupId>
    <artifactId>gmark</artifactId>
    <version>1.2</version>
</dependency>
```

[1] https://github.com/RoanH/gMark
[2] https://hub.docker.com/r/roanh/gmark
[3] https://mvnrepository.com/artifact/dev.roanh.gmark/gmark
[4] https://gmark.docs.roanh.dev/

B.3 Development of gMark

The gMark repository contains an Eclipse[5] and a Gradle[6] project with Util[7] and Apache Commons CLI[8] as the only dependencies. Development work can be done using the Eclipse IDE or using any other Gradle compatible IDE. Unit testing is employed to test core functionality and continuous integration is used to run checks on the source files, check for regressions using the unit tests, and to generate release publications. Compiling the runnable Java archive (JAR) release of gMark using Gradle can be done using the following command in the gMark directory:

```
./gradlew clientJar
```

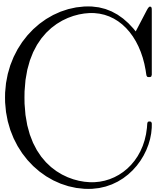
After running this command the generated JAR can be found in the `build/libs` directory. On Windows `./gradlew.bat` should be used instead of `./gradlew`. Note that this command builds gMark including its graphical user interface, if only command line usage is desired then a slightly smaller release JAR can be obtained using:

```
./gradlew cliJar
```

Again, the generated JAR can be found in the `build/libs` directory and on Windows `./gradlew.bat` should be used instead of `./gradlew`.

[5] `https://www.eclipse.org/`
[6] `https://gradle.org/`
[7] `https://github.com/RoanH/Util`
[8] `https://commons.apache.org/proper/commons-cli/introduction.html`

Index Setup



This appendix will detail how to use and obtain a copy of the *CPQ*-native Index software. The software is released on GitHub[1] and the README file of the repository will always contain the most up-to-date instructions on using the software. This appendix is written for release 1.0 of the *CPQ*-native Index and may not be fully correct for newer releases.

To support a wide variety of use cases the index software is a available in a number of different formats:

1) As a standalone executable with a command line interface.

2) As a docker image.

3) As a maven artifact.

4) For development.

However, it should be noted that some of these options provide less control over the execution environment than others. The recommended way to use the index software is to either incorporate it directly in some other piece of software, or to use the Java archive release if no customisation is required. Allocating as much RAM to the heap of the process as possible is recommended when computing an index for large graphs.

C.1 Command-line Usage

When using the command line interface of the index, the following arguments are supported:

```
usage: index [-c] -d <file> [-f] [-h] [-i <max>] -k <k> [-l] -o <file> [-t <number>] [-v <file>]
 -c,--cores                  If passed then cores will be computed.
 -d,--data <file>            The graph file to create an index for or a saved index file.
 -f,--full                   If passed the saved index has all information required to compute cores.
 -h,--help                   Prints this help text.
 -i,--intersections <max>    The maximum number of branches for intersection cores (default unlimited).
 -k,--diameter <k>           The value of k (diameter) to compute the index for.
 -l,--labels                 If passed then labels will be computed.
 -o,--output <file>          The file to save the constructed index to.
 -t,--threads <number>       The number of threads to use for core computation (1 by default).
 -v,--verbose <file>         Turns on verbose logging, optionally to a file or Discord.
```

For example, a base index without cores can be constructed using:

```
java -Xmx1900G -jar Index.jar -f -d graph.edge -k 2 -t 64 -v log.txt -o base_index.idx
```

The `-Xmx` argument is passed to Java and controls the maximum amount of RAM that will be used for the heap. Later a version of the base index with cores can be constructed using:

```
java -Xmx1900G -jar Index.jar -d base_index.idx -k 2 -c -t 64 -v discord:log.txt -o index.idx
```

Note that `discord:` can be prepended to the log file argument, which will send computation progress updates to the webhook configured in the `DISCORD_WEBHOOK` variable in the `Main` class of the program. By default no webhook is configured, so configuring this requires compiling from source as descried in Section C.5. For testing the robots dataset is available in the CPQ-aware Index repository[2].

[1] `https://github.com/RoanH/CPQ-native-index`
[2] `https://github.com/yuya-s/CPQ-aware-index/blob/main/data/robots.edge`

C.2 Executable Download

Pre-compiled binaries of this version of the *CPQ*-native Index for all major operating systems can be found in the repository README and releases section[3]. The current version of the index requires Java 17 or higher to run. Note that the Windows executable release does not offer the same degree of control over the heap size as the Java archive version.

C.3 Docker Image

The *CPQ*-native Index is available as a docker image[4] on Docker Hub. This means that you can obtain the image using the following command:

```
docker pull roanh/cpq-native-index:latest
```

Using the image then works much the same as regular command line usage as described in Section C.1. For example, we can generate the example base index from Section C.1 using the following command:

```
docker run --rm -v "$PWD/data:/data" roanh/cpq-native-index:latest java -Xmx1900G -jar
    Index.jar -f -d /data/graph.edge -k 2 -t 64 -v /data/log.txt -o /data/base_index.idx
```

Note that we mount a local folder called 'data' into the container to pass our graph and to retrieve the generated index and logs. Also note that the entry point of the image has to be overridden to explicitly allocate more RAM to the heap.

C.4 Maven Artifact

The *CPQ*-native Index is available on maven central as an artifact[5], this way it can be included directly in another Java project using build tools like Gradle and Maven. Using this method it also becomes possible to directly use all the implemented constructs and utilities. A hosted version of the javadoc for for the index can be found at cpqnativeindex.docs.roanh.dev[6].

Listing C.1: Gradle setup for *CPQ*-native Index.

```
repositories{
    mavenCentral()
}

dependencies{
    implementation 'dev.roanh.cpqnativeindex:cpq-native-index:1.0'
}
```

Listing C.2: Maven setup for the *CPQ*-native Index.

```
<dependency>
    <groupId>dev.roanh.cpqnativeindex</groupId>
    <artifactId>cpq-native-index</artifactId>
    <version>1.0</version>
</dependency>
```

Note that the maven release of the index does include pre-compiled binaries of the native library for Windows and Linux. These binaries will be extracted to the `lib` folder when used. However, it is possible that directly compiling on the target system may result in better performance.

[3] https://github.com/RoanH/CPQ-native-index/releases
[4] https://hub.docker.com/r/roanh/cpq-native-index
[5] https://mvnrepository.com/artifact/dev.roanh.cpqnativeindex/cpq-native-index
[6] https://cpqnativeindex.docs.roanh.dev/

C.5 Development of the Index

The *CPQ*-native Index repository contains an Eclipse[7] and a Gradle[8] project with gMark[9] and Apache Commons CLI[10] as the only dependencies. In addition, in order to compile the native library a C compiler and CMake are also required. Development work can be done using the Eclipse IDE or using any other Gradle compatible IDE. Unit testing is employed to test core functionality. Continuous integration is used to run checks on the source files, check for regressions using the unit tests, and to generate release publications.

Compiling the native library can be done using the following command in the CPQ-native Index directory:

```
./gradlew compileNatives
```

Running this command will place the compiled native library in the `lib` directory. Next, compiling the runnable Java archive (JAR) release of the index using Gradle can be done by running the following command in the same directory:

```
./gradlew shadowJar
```

After running this command the generated JAR can be found in the `build/libs` directory. On windows `./gradlew.bat` should be used for both commands instead of `./gradlew`. Also note that the native libraries should always be compiled before building a complete release JAR.

In software, an index can be constructed using the constructor in Listing C.3.

Listing C.3: Software index construction.

```
Index index = new Index(
    IndexUtil.readGraph(graphFile),
    k,
    cores,
    labels,
    threads,
    intersections,
    listener
);
```

[7] https://www.eclipse.org/
[8] https://gradle.org/
[9] https://github.com/RoanH/gMark
[10] https://commons.apache.org/proper/commons-cli/introduction.html