

MASTER'S THESIS 2025

# An Evaluation of Approaches to Code Formatting

Simon Thuresson, Tomas Nyberg

Elektroteknik  
Datateknik

ISSN 1650-2884

LU-CS-EX: 2025-10

DEPARTMENT OF COMPUTER SCIENCE

LTH | LUND UNIVERSITY





EXAMENSARBETE  
Datavetenskap

LU-CS-EX: 2025-10

**An Evaluation of Approaches to Code  
Formatting**

En utvärdering av metoder för  
kodformatering

Simon Thuresson, Tomas Nyberg



---

# An Evaluation of Approaches to Code Formatting

(A Case Study on the Cypher Query Language)

---

Simon Thuresson  
sthuresson00@gmail.com

Tomas Nyberg  
tomasgnyberg@gmail.com

May 20, 2025

Master's thesis work carried out at Neo4j, Inc.

Supervisors: Emma Söderberg, [emma.soderberg@cs.lth.se](mailto:emma.soderberg@cs.lth.se)  
Oskar Damkjær, [oskar.damkjaer@neo4j.com](mailto:oskar.damkjaer@neo4j.com)

Examiner: Görel Hedin, [gorel.hedin@cs.lth.se](mailto:gorel.hedin@cs.lth.se)



## Abstract

Code formatters are software development tools used to improve readability and enforce consistent styling through whitespace modifications. Since they can be implemented using various approaches, choosing the most suitable one can be challenging. In this thesis, we explore three approaches to formatting—pretty printing, optimization-based, and layout-based—and evaluate each approach by using it to implement a formatter for the Cypher query language. We compare all three implementations in terms of output quality, performance, and maintainability, and make them publicly available through an open-source repository, with the final implementation actively running in production. Additionally, we propose improvements to the formatting rule set for Cypher to address incompleteness and ambiguity. Our findings conclusively demonstrate that the layout-based approach yields the best results in these dimensions for Cypher, indicating it could also be preferable for other languages, provided their styling conventions do not prevent its use.

**Keywords:** Code formatting, code style, Cypher, formatting algorithms, ANTLR.



# Acknowledgements

---

We would like to express our gratitude to our supervisor, Dr. Emma Söderberg, for her invaluable guidance and feedback, which has been instrumental in achieving the goals of this thesis. Special thanks also go to Oskar Damkjær for his many insightful ideas and contributions, which helped shape the direction of the project. We are likewise grateful for the generous support we received from Nacho Cerdón Castillo with regard to the implementation of the Cypher formatter. We would also like to thank Professor Görel Hedin for acting as the examiner for this thesis.

Lastly, we would like to thank everyone at Neo4j who answered our survey and provided us with valuable feedback, and Neo4j as an organization for giving us the opportunity and resources to undertake this project.



# Contents

---

|          |                                            |           |
|----------|--------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                        | <b>9</b>  |
| 1.1      | Study Context–Neo4j and Cypher . . . . .   | 10        |
| 1.1.1    | Cypher Style Guide . . . . .               | 10        |
| 1.2      | Research Questions . . . . .               | 12        |
| 1.3      | Related Work . . . . .                     | 12        |
| 1.4      | Scientific Contribution . . . . .          | 13        |
| 1.5      | Contribution Statement . . . . .           | 13        |
| <b>2</b> | <b>Background</b>                          | <b>15</b> |
| 2.1      | Lexers, Parsers, ASTs, and ANTLR . . . . . | 15        |
| 2.1.1    | Lexing . . . . .                           | 15        |
| 2.1.2    | Parsing . . . . .                          | 16        |
| 2.1.3    | ANTLR . . . . .                            | 16        |
| 2.2      | Formatting Terminology . . . . .           | 17        |
| 2.2.1    | Pretty Printers and Formatters . . . . .   | 17        |
| 2.2.2    | Chunk . . . . .                            | 17        |
| 2.2.3    | Group . . . . .                            | 18        |
| 2.3      | Existing Formatting Approaches . . . . .   | 18        |
| 2.3.1    | Pretty Printing . . . . .                  | 18        |
| 2.3.2    | Optimization-based . . . . .               | 19        |
| 2.3.3    | Layout-based Approaches . . . . .          | 19        |
| 2.4      | Other Code Formatters . . . . .            | 21        |
| 2.4.1    | Prettier (JavaScript) . . . . .            | 21        |
| 2.4.2    | Black (Python) . . . . .                   | 22        |
| 2.4.3    | SQL formatter . . . . .                    | 23        |
| <b>3</b> | <b>Methodology</b>                         | <b>25</b> |
| 3.1      | Data Collection . . . . .                  | 25        |
| 3.1.1    | Cypher Style Guide . . . . .               | 25        |
| 3.1.2    | User Feedback in Slack . . . . .           | 27        |

---

|                               |                                                |           |
|-------------------------------|------------------------------------------------|-----------|
| 3.1.3                         | User Survey . . . . .                          | 27        |
| 3.1.4                         | Benchmark . . . . .                            | 28        |
| 3.2                           | Data Analysis . . . . .                        | 28        |
| 3.2.1                         | Extract Relevant Rules . . . . .               | 29        |
| 3.2.2                         | Filter User Feedback . . . . .                 | 29        |
| 3.2.3                         | Compare Code Styles . . . . .                  | 29        |
| 3.2.4                         | Performance Analysis . . . . .                 | 30        |
| 3.3                           | Implementation . . . . .                       | 30        |
| 3.4                           | Testing and Verification . . . . .             | 30        |
| 3.4.1                         | Testing Strategy . . . . .                     | 30        |
| 3.4.2                         | Essential Formatter Properties . . . . .       | 30        |
| 3.4.3                         | Verification at Scale . . . . .                | 31        |
| <b>4</b>                      | <b>Formatting Rule Set for Cypher</b>          | <b>33</b> |
| 4.1                           | Scope of a Formatter . . . . .                 | 33        |
| 4.2                           | Survey Results and User Feedback . . . . .     | 35        |
| 4.3                           | Added Rules . . . . .                          | 36        |
| <b>5</b>                      | <b>Technical Design</b>                        | <b>43</b> |
| 5.1                           | Pretty Printer Approach . . . . .              | 43        |
| 5.1.1                         | Formatting Example . . . . .                   | 44        |
| 5.2                           | Optimization-based Approach . . . . .          | 44        |
| 5.2.1                         | High-level Overview . . . . .                  | 44        |
| 5.2.2                         | Data Structures . . . . .                      | 46        |
| 5.2.3                         | Best-first Solution Search . . . . .           | 49        |
| 5.2.4                         | Dynamic Programming . . . . .                  | 50        |
| 5.2.5                         | Alignment and Indentation . . . . .            | 54        |
| 5.2.6                         | Formatting Example . . . . .                   | 55        |
| 5.3                           | Layout-based Approach . . . . .                | 55        |
| 5.3.1                         | Determining Group Split . . . . .              | 57        |
| 5.3.2                         | Separation of Indentation and Groups . . . . . | 57        |
| 5.3.3                         | Layout Engine Variants . . . . .               | 58        |
| 5.3.4                         | Formatting Example . . . . .                   | 58        |
| <b>6</b>                      | <b>Evaluation of Approaches</b>                | <b>61</b> |
| 6.1                           | Output Quality . . . . .                       | 62        |
| 6.2                           | Performance . . . . .                          | 62        |
| 6.3                           | Maintainability . . . . .                      | 63        |
| <b>7</b>                      | <b>Limitations</b>                             | <b>65</b> |
| 7.1                           | Styling Preferences . . . . .                  | 65        |
| 7.2                           | Implementations of Approaches . . . . .        | 66        |
| <b>8</b>                      | <b>Conclusions</b>                             | <b>67</b> |
| 8.1                           | Future Work . . . . .                          | 67        |
| <b>Appendix A User Survey</b> |                                                | <b>75</b> |

|                                  |    |
|----------------------------------|----|
| Appendix B Generative AI Prompts | 81 |
|----------------------------------|----|



# Chapter 1

## Introduction

---

Code formatters are used extensively throughout the software industry and their use is often considered a prerequisite before submitting code for review [42]. These tools automatically make whitespace modifications, ensuring that the code complies with styling preferences without changing its semantic meaning. Examples of such whitespace modifications could be indenting code that is logically related or introducing line breaks to ensure that all code remains within a maximum column width. The goal of these superficial modifications is generally to improve readability [35] and to enforce uniform styling throughout the codebase [24]. Research on formatting and readability by Baecker [7] found that consistently formatted code improves readability by 25%, and readability is recognized as a key factor in a maintainable codebase [47]. Furthermore, lower readability has also been identified by researchers as correlated with a higher probability of bugs [8].

There exist multiple viable approaches to implement a formatter that can apply a set of formatting rules [25, 34, 16, 50]. Our research indicates that the implementation approaches of formatters can broadly be divided into three main categories: pretty printing, optimization-based, and layout-based approaches, although additional approaches may exist. In this thesis, we implement a formatter using each approach and evaluate them in relation to each other in terms of output quality, performance, and maintainability. These approaches can vary greatly in multiple aspects and, as will be explored, choosing the right approach is of significant importance when implementing a formatter.

The target language for which we will implement a formatter is the Cypher query language, which is a declarative query language used to express queries against a graph database. Readability is, in our view, especially important for query languages, which are generally declarative and can capture a significant amount of logic in relatively few lines of code. Consequently, Neo4j, the company behind Cypher, provides a style guide [31] with recommendations on how the language should be formatted. However, there exists no automated tooling to format the language according to the aforementioned styling recommendations. In addition, the style guide does not offer a comprehensive set of rules; it is at times incomplete, ambiguous, or self-contradictory. This has led to inconsistently formatted code in internal

Neo4j codebases, as seen, for example, in the official repository for the database [32], as well as in the official documentation for the language [29]. The aim of this thesis is to provide a comprehensive set of rules for formatting Cypher in addition to introducing a formatter that automatically applies these rules to any query.

Although Cypher is a query language, not a general-purpose one, we argue that there are sufficient similarities to allow us to draw conclusions for a general-purpose language formatter as well. In Cypher, common language constructs such as lists, maps, and binary expressions closely resemble the ones found in general-purpose languages. Similarly, there is often a need for dynamic line breaks for lines exceeding a maximum column width.

## 1.1 Study Context—Neo4j and Cypher

Neo4j is currently the most widely used graph database [10], designed to overcome the limitations of traditional relational databases when handling highly connected data. Through the use of a graph-native data model, it enables highly efficient querying of connected data, a scenario in which relational databases typically struggle [27].

To effectively express complex graph queries, Neo4j developed Cypher, a declarative query language. The language is designed to provide a more intuitive way of expressing graph queries with a pattern-matching syntax that resembles the actual relationships in the underlying graph. Compared to other languages, it is perhaps most similar to Structured Query Language (SQL), with keywords using capital letters and clear separation of clauses. Much like SQL, it is also highly declarative, allowing the user to formulate the logical query without having to think about the underlying details of how it is going to be executed [30]. Listing 1 shows a simple example of querying for related nodes (equivalent to rows in a relational database) based on certain criteria. A similar query in SQL would involve complex joins between different tables, and thus would almost certainly be less performant and less readable [33].

```
1 MATCH (user:Person {name: "Alice"})-[:FRIEND_OF]->(friend)
2 RETURN friend.name;
```

**Listing 1:** Cypher query to retrieve all friends of Alice

### 1.1.1 Cypher Style Guide

The Cypher style guide provides guidelines for writing queries with the goal of making them as easy to read as possible [31]. Most of its recommendations revolve around whitespace and thus fall within the scope of a formatter. An example of applying the recommendations can be seen in Listing 2, where different clauses have been split into separate lines, and all keywords have been capitalized.

Although the style guide lists many recommendations, it is by no means comprehensive. The Cypher language contains a vast number of different constructs, and many of them do

```

1 // Does not follow the style guide recommendations
2 match (n) where n.name contains "s" return n.name
3
4 // Follows the style guide recommendations ("Start a new clause on a
5 new line" and "Write keywords in upper case")
6 MATCH (n)
7 WHERE n.name CONTAINS "s"
8 RETURN n.name

```

**Listing 2:** Cypher code that does not conform to the style guide compared to code that does

not have explicit recommendations for how they should be formatted. It also provides very few recommendations on how to split statements that become too long to fit within a maximum column width, leaving those cases as an exercise to the reader. Based on our observations of Cypher queries in public codebases, most users, when faced with a case where no convention exists, adopt conventions commonly found in SQL instead. An example of this would be long logical OR/AND expressions, which we found were most commonly formatted similarly to what is shown in Listing 3. This style is frequently found in SQL code, and given that Cypher and SQL are relatively similar, the style preferences seem to transfer well between them.

```

1 // First variant: OR to the left
2 WHERE p.name = "Alberta"
3     OR p.name = "Berta"
4     OR p.name = "C"
5     OR p.age > 30
6
7 // Second variant: OR to the right
8 WHERE
9     p.name = "Alberta" OR
10    p.name = "Berta" OR
11    p.name = "C" OR
12    p.position = "Manager"

```

**Listing 3:** Example of how users commonly formatted long OR expressions (no recommendation is provided by the style guide)

## 1.2 Research Questions

In order to achieve our goals of establishing a comprehensive set of formatting rules and evaluating different formatting approaches, we aim to answer the following research questions:

**RQ<sub>1</sub>:** How should the set of formatting rules (style guide) for the Cypher query language be defined?

Although our primary goal is to evaluate different formatting approaches, establishing a comprehensive set of formatting rules for Cypher is necessary to ensure a fair comparison for RQ2.

**RQ<sub>2</sub>:** How do existing approaches to code formatting compare with respect to output quality, performance, and maintainability?

The motivation behind choosing output quality, performance, and maintainability as our metrics for RQ2 is that we consider them to provide a holistic view of the quality of a formatter. As a user of a formatter, output quality and performance are the most critical elements; if either do not meet acceptable standards, the formatter is unlikely to be used. Maintainability complements these aspects by evaluating the implementation complexity, giving insight into the practicality of each approach.

## 1.3 Related Work

Among the literature reviewed, we identified three sources that are the most relevant for this thesis [34, 50, 16]. However, none of them address any research question comparable to our RQ1. Instead, they seem to apply predefined rules, probably because the formatting guidelines for the respective languages had already been established. With regard to which implementation approach should be applied, the three reviewed papers adopt two different approaches. Both Geirsson [16] and Nystrom [34] follow a similar optimization-based approach, while Wadler [50] adopts a layout-based approach.

With respect to RQ2, all three sources emphasize performance. This likely reflects the fact that poor performance can render a formatter unsuitable for practical use, thereby justifying the strong emphasis placed on it. Output quality emerges as a common theme throughout the sources, though none address it consistently in depth, likely because they follow their respective established styling conventions. Maintainability appears to be out of the scope of these sources. However, Geirsson [16] appears to place a secondary focus on using Scala native code, suggesting an indirect consideration for increased maintainability.

None of the three main papers, nor any other we reviewed, address how to implement a formatter specifically for a query language. To our knowledge, no prior research has focused specifically on this topic. Although we explore formatting approaches in a general sense, our thesis appears to be uniquely positioned among the formatting literature as it directly addresses the formatting of a query language.

## 1.4 Scientific Contribution

The primary contribution of this thesis is the exploration of different approaches to the implementation of a formatter. We evaluate each approach by their output quality, performance, and maintainability, providing valuable guidelines for future formatter development. Additionally, we present a comprehensive set of formatting rules for the Cypher query language that ties into the choice of implementation approach. As a practical outcome, we provide open-source implementations for each approach explored, with the final implementation being a complete Cypher formatter currently used in production.

## 1.5 Contribution Statement

We contributed equally to the design and implementation of the Cypher formatter and to writing this thesis. By always reviewing each other's work, both code and text, we ensured equal contributions to all parts of this thesis.

We used Generative AI for two purposes. First, to rephrase sentences and second, to explore ideas while developing the formatter. As for our thesis, everything has been written by us, and therefore all arguments and ideas are our own. To enhance readability, we have prompted Generative AI for rephrased sentences that improve flow and accuracy without altering semantic meaning. For implementing our formatter, Generative AI has been used to aid development by, for example, doing refactorization or suggesting names for function and variables. We include an excerpt in Appendix B of prompts we have used.



# Chapter 2

## Background

---

In this chapter, we provide the background necessary to understand the broader topic of formatting. We cover the theoretical background from compiler theory, along with commonly used terminology within formatters. Additionally, we cover the predominant approaches to implementing a modern code formatter, in particular the approaches we aim to implement. Finally, we provide a brief discussion of formatters for other languages and their influence on this thesis.

### 2.1 Lexers, Parsers, ASTs, and ANTLR

When writing software that processes source code, it is common to employ compiler techniques [1]. In the context of code formatting, the compiler techniques that are most relevant are **lexing** and **parsing**. Together, they allow us to take a string written in a programming language and convert it into another intermediate representation that makes it easier to process. The most common representation used in this context is the Abstract Syntax Tree (AST), or its more detailed form, the Concrete Syntax Tree (CST). An AST describes the parsed input in a structured way that represents the logical structure of the source code. Similarly, a CST contains all the information that the AST does but additionally retains all syntactic trivia such as whitespace and comments. This can be helpful, for example, for a formatter that wants to retain whitespace from the original input.

#### 2.1.1 Lexing

Lexing is typically the first step in compilers and other programs that process source code [1]. This step takes the raw source code (a string of characters) and converts it into a stream of **tokens**. Tokens have a type and a value, and can be considered atomic units of languages, analogous to words in a natural language. For a simple example of what the lexing process looks like, specifically in the context of Cypher, consider the source code and the parsed

tokens in Listing 4. Note that each token has a unique type, which distinguishes it from

```
1 // Cypher source code
2 MATCH (n:Person)
3 // Lexer output
4 [{type: keyword, text: 'MATCH'}, {type:PAREN, text: '('},
  {type:identifier, text: 'n'}, {type:label, text: ':Person'},
  {type:PAREN, text: ')'}]
```

**Listing 4:** Lexing example

other tokens that contain the same text but describe something else. This allows us to, for instance, treat parentheses appearing in node patterns differently from those appearing in string literals, as illustrated by how the node parentheses in Listing 4 have the type **PAREN**.

## 2.1.2 Parsing

Once the input has been converted into a series of tokens, the tokens can be assembled into another form that represents their structure based on the grammar of the language. The grammar, much like in natural languages, describes the valid ways in which a string of that language can be constructed. Consider, for instance, the grammar rule for a Cypher **MATCH** clause, described in Listing 5. This grammar rule requires some constructs to be present,

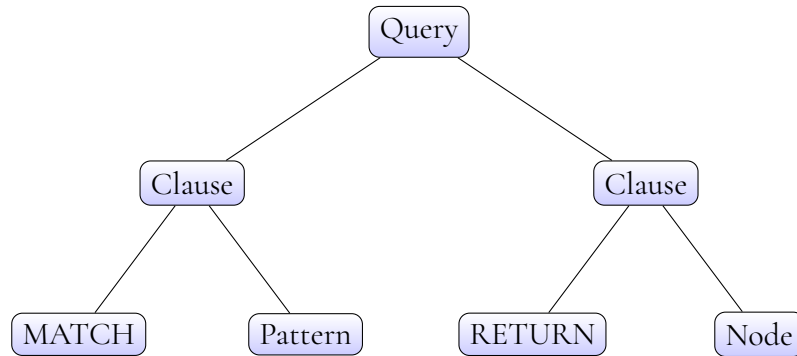
```
1 matchClause
2   : MATCH pattern hint* whereClause?
3   ;
```

**Listing 5:** Grammar that describes a Cypher match clause (slightly simplified)

such as **MATCH**, while the **whereClause** is optional (denoted by '?'), and the **hint** expression may appear multiple times (denoted by '\*'). If the input string contains tokens that do not conform to the grammar rules specified for the language, it is considered syntactically invalid. The output of the parsing step is typically a tree that represents the hierarchical structure of the program. In our case, the output will be the CST that is later used by the formatter.

## 2.1.3 ANTLR

Doing the lexing and parsing steps manually can be time consuming and error prone. Fortunately, engineers have not had to do so since the 1970s, when compiler compilers, commonly referred to as parser generators, became popular. These parser generators allow the user to simply provide the lexical and grammatical rules for their language, and the tool then



**Figure 2.1:** A simplified AST for the Cypher query `MATCH (n) RETURN n`.

automatically generates source code that implements both a lexer and a parser for that language [22]. One such tool is ANTLR (ANOther Tool for Language Recognition) [38], and together with the Cypher grammar that the openCypher specification provides [36], we can automatically generate a program that parses Cypher into a CST.

In addition to generating a parser, ANTLR also provides helpful skeletons for classes used to process the resulting CST. Two such classes are **visitors** and **listeners**. A listener enables running code each time a certain type of node is encountered, and a visitor enables explicit control over the tree traversal for specific nodes in the CST. The visitor pattern can be used to implement a formatter, for example, by implementing visit methods for all language constructs that need to be formatted in a certain way [34].

## 2.2 Formatting Terminology

This thesis has adopted a combination of terms and concepts inspired by previous work on formatting. In this section, we cover the most relevant terms and concepts that are necessary to understand upcoming chapters.

### 2.2.1 Pretty Printers and Formatters

The terms "pretty printer" and "formatter" both refer to software that makes whitespace modifications with the goal of increased readability. Although these terms are occasionally used as synonyms, we prefer to make the distinction that a pretty printer is a subset of a formatter. Specifically, it is a formatter that does not support dynamic line breaks and, therefore, does not try to keep each line within a maximum column width.

### 2.2.2 Chunk

A fundamental concept used in the formatting context is that of a **chunk** [34, 50, 19]. A **chunk** is the atomic unit in formatting, consisting of characters that should never split. The term **chunk** was adopted from **dartfmt** [34], but similar ideas appear in other papers on formatting [19, 50] under the name **Text**.

### 2.2.3 Group

Another key term, **group**, was introduced by Wadler [50]. A **group** consists of one or more **chunks**, and either splits between all of its **chunks**, or does not split at all. The **dartfmt** [34] formatter contains a similar concept called **rule**. However, a key difference between a **rule** and a **group** is that **rule** allows for a wider variety of splits. A **rule** can for instance allow two function arguments to remain on the same line while only the third argument is split to a new line. Another similar concept is that of a **policy**, found in **scalafmt** [16] and described as: *"A Policy is an enforced formatting layout over a region."*, where a region would be comparable to a group encompassing chunks.

## 2.3 Existing Formatting Approaches

There are several ways to implement a formatter, and choosing the right approach is key to achieving the desired results. In this section, we outline the three approaches we found to be the most common among formatters with widespread adoption today [5, 34, 16, 25, 6]. Each of these approaches results in the same output in the case where every line to be formatted fits within the preferred maximum column width. However, differences arise when newlines need to be inserted, at which point the output can start to vary drastically.

### 2.3.1 Pretty Printing

As described in Section 2.2.1, we define a pretty printer as a formatter without support for dynamic line breaks. A simple example is a pretty printer for JavaScript Object Notation (JSON) that converts code as shown in Listing 6. This type of formatter is, in our opinion,

```
1 // Before pretty printing
2 {"name": "Alice", "age": 30, "email": "alice@example.com", "hobbies":
3  ["reading", "hiking", "coding"]}
4 // After pretty printing
5 {
6     "name": "Alice",
7     "age": 30,
8     "email": "alice@example.com",
9     "hobbies": [
10         "reading",
11         "hiking",
12         "coding"
13     ]
14 }
```

**Listing 6:** Example of "pretty printing" JSON

straightforward to implement, especially if parser-generating tools such as the ones described

in Section 2.1 are used. Only language constructs that require special formatting, such as breaking the line and adding indentation after an opening brace, need dedicated handling. Because of this simplicity, these formatters can be very efficient and easy to maintain. As a result, the pretty printing approach often serves as the foundation for more advanced formatter implementations that support dynamic line breaks [34].

The main disadvantage of pretty printing, as opposed to a more advanced approach, is that it does not attempt to keep the code within a maximum column width. If long keys were used, for instance, the aforementioned JSON pretty printer would happily let the code overflow smaller screen widths. In the case of JSON, that is not much of an issue, but for most other popular languages it can significantly decrease readability. Based on the popularity of formatters that automatically break long lines, as opposed to ones that do not, this easily becomes a deal-breaker.

It should be noted that the Go formatter `gofmt` is essentially just a pretty printer, but despite this, it is highly popular [17]. Instead, users seem to have adopted the mindset that long lines should simply be rewritten so that they fit [48].

### 2.3.2 Optimization-based

Optimization-based formatters are those that perform a search in the space of all possible formatting layouts and try to find the optimal one. Between every pair of tokens, the formatter is given an option of what kind of whitespace it wants to insert between them, most commonly a newline or a space. These choices are then ranked according to some form of heuristic, often referred to as a "cost", and the objective becomes to find the whitespace choices that result in the lowest cumulative cost. The formatter faces two choices between every token, and each choice affects future choices, resulting in a decision tree that grows exponentially. In order to make the problem tractable, this type of formatter typically employs a variety of ad hoc optimizations.

The approach outlined above was inspired by the implementations of the formatters `clang-fmt` [25], `dartfmt` [34], and `scalafmt` [16], all three of which perform an optimized search for the best formatting solution. The optimizations they use are typically specific to their target language and, therefore, their implementations can vary significantly. The advantage of this approach is that it is very flexible in terms of what output it enables; the cost can be defined freely, enabling the author to define unique types of styling. An example of this is the multitude of ways that `dartfmt` can split function calls, as illustrated in Listing 7. The downside is that the taming of the exponential search space becomes incredibly complex. This can easily lead to convoluted implementations that require makeshift tooling to effectively debug [16], and achieving the intended formatting output can be very challenging [34].

### 2.3.3 Layout-based Approaches

The layout-based approach to formatting fits somewhere between the rigidity of pretty printers and the convoluted search strategies of optimization-based formatters. Layout-based formatting entails transforming the input into an abstract representation composed of a series of "documents", joined by different operators. Depending on whether or not a document fits within the available column width, it decides to render itself either in "flat" or "broken" form.

```
1 // 0: Don't split argument list at all.
2 function(first, second, third)
3
4 // 1: Split before the first argument.
5 function(
6     first, second, third)
7
8 // 2: Split only before the last argument.
9 function(first, second,
10         third)
11
12 // 3: Split only before the middle argument.
13 function(first,
14         second, third)
15
16 // 4: Split before all of them.
17 function(
18     first,
19     second,
20     third)
```

**Listing 7:** The different ways `dartfmt` can split a Dart function call, ordered by increasing levels of cost.

This process is then repeated for each subdocument of that document. In practice, this essentially means deciding whether to put spaces between each subdocument (flat), or to add indentation around itself and insert newlines between each subdocument (broken). This ensures that language constructs that fit on one line are printed compactly, and otherwise they are recursively split until they can no longer split or fit within the available space [37, 19, 50].

The relatively rigid rules of layout-based formatting make the formatter very efficient, and simpler implementations often run in linear time [37, 50]. Despite this simplicity, or perhaps even because of it, the output these formatters produce can be very desirable. **Prettier**, the JavaScript formatter, is based on this approach [5], and is one of the most popular code formatters in history [2]. Recent research has also shown that it is possible to increase the expressiveness of these types of formatters to allow for more styling choices [39].

## 2.4 Other Code Formatters

Formatting tools have become ubiquitous in development environments today, indicating that they play an important role in modern software development [7]. As a result, most popular programming languages provide formatting tools to their users, and these generally enjoy widespread usage. JavaScript has **Prettier** [3], Python has tools like **Black** [15] and **autopep8** [18] (pep8 being the official style guide for Python), Go has **gofmt** [6], C/C++ have **clang-format** [40], SQL has **SQL formatter** [43], and so on.

Although much can be said about these formatting tools, this section will focus on two things: how **Prettier** and **Black** have influenced our philosophy of what a formatter should be, and the styling choices taken in **SQL formatter**, one of the most popular formatters available for a query language [20].

### 2.4.1 Prettier (JavaScript)

**Prettier** [3] is the de facto standard formatter for JavaScript and TypeScript, and can also be used for other languages such as Cascading Style Sheets (CSS) [23]. Its name, as well as much of its implementation and styling choices [5] are derived from the seminal formatting paper by Wadler [50]. In terms of the categorization provided in Section 2.3, it is categorized as a layout-based formatter. **Prettier** is opinionated and offers minimal configuration options and, in fact, it intentionally does not support any kind of global configuration. In their words:

*"This is to make sure that when a project is copied to another computer, Prettier's behavior stays the same. Otherwise, Prettier wouldn't be able to guarantee that everybody in a team gets the same consistent results." (Prettier authors, [4])*

This preference for opinionated styling over allowing configuration seems to have become somewhat of a tradition among code formatters, one that we intend to uphold.

In JavaScript, it is common to have large groups of text that cannot fit on one line. These groups could, for instance, be function arguments, long logical expressions, or list literals. **Prettier**, like most industry-standard formatters, automatically breaks lines that exceed a maximum column width (usually 80 characters). Consider the list literal shown in Listing 8 (taken from the implementation of our Cypher formatter and formatted by **Prettier**). If

```
1 const trailingCharacters = [  
2   CypherCmdLexer.SEMICOLON,  
3   CypherCmdLexer.COMMA,  
4   CypherCmdLexer.COLON,  
5   CypherCmdLexer.RPAREN,  
6   CypherCmdLexer.RBRACKET,  
7 ];
```

**Listing 8:** JavaScript list formatted in the "Prettier" fashion

this list literal were written in a single line, the `trailingCharacters` list would exceed 100 characters, exceeding the widely accepted maximum of 80 characters [49, 14]. The layout-based approach to breaking lines when formatting, as illustrated in the aforementioned listing, is to introduce a line break between every element in the list, after the commas.

## 2.4.2 Black (Python)

*"Any color you like." [15]*

**Black** [15] is a Python formatter with widespread adoption, used by notable Python-based projects such as **Django** [12] and **pytest** [41]. The primary adjective used by its maintainers to describe it is "uncompromising", and this is by design. It takes the philosophy of minimal configuration to its extreme and provides even fewer configuration options than **Prettier** [15].

Python formatters have to contend with syntactically significant whitespace and are thus not able to offer us much in terms of inspiration for styling. But what the **Black** formatter can provide us with is a good philosophy of what a formatter should be. In particular, its uncompromising stance when it comes to code formatting should not be seen as regressive or as a bug, but rather as a feature. If a formatter provides minimal configuration, it means that all projects that use it will be formatted the same way, and code changes within the same repository will not contain unnecessary whitespace changes. In addition, it makes it easier for any developer familiar with the style to start contributing to another project that uses it, and they do not have to concern themselves with such minutiae. As the authors of **Black** put it:

*"You will save time and mental energy for more important matters. Blackened code looks the same regardless of the project you're reading. Formatting becomes transparent after a while and you can focus on the content instead." (Black authors, [15])*

The Cypher formatter described in this thesis will follow the philosophy described by **Black** and will likewise aim to be uncompromising by design. One way this will become immediately obvious to our users is the fact that all keywords are automatically converted to uppercase, and this is not configurable.

### 2.4.3 SQL formatter

**SQL formatter** [43] is the most popular formatter for SQL [21] in terms of weekly downloads [9] and is also the most popular formatter for a query language. SQL is very similar to Cypher; they are both query languages and share many of the same language constructs. Because of this, formatting conventions in SQL provide a useful comparison point for Cypher formatting. Although **SQL formatter** does not fit neatly into any of the categories listed in Section 2.3, its output closely resembles that of the layout-based approach. Its authors have also noted that its current architecture has "*fundamental problems*" [43] and, consequently, they have written another version that explicitly follows the Prettier algorithm of Wadler [43, 50].

An example of code formatted by **SQL formatter** can be seen in Listing 9. Notably,

```
1 SELECT
2     e.employee_id,
3     e.first_name,
4     e.last_name,
5     ad.region,
6     ad.total_sales,
7     ad.total_adjusted_sales,
8     CASE
9         WHEN ad.avg_adjusted_sales > 10000 THEN 'High Performer'
10        WHEN ad.avg_adjusted_sales >= 5000
11            AND ad.avg_adjusted_sales <= 10000 THEN 'Average Performer'
12        ELSE 'Low Performer'
13    END AS performance_category
14 FROM
15     employees e
```

Listing 9: SQL formatted by **SQL formatter**

the query is split between the two clauses (**SELECT** and **FROM**) and the long list of items below the **SELECT** is split between each item and indented by two spaces. Although no explicit convention exists for formatting long groups in Cypher, it is common (based on our observations of sample queries) to follow the same principle when formatting Cypher as well. To illustrate this principle applied to Cypher, consider the Cypher query shown in Listing 10. In this case, a newline along with two indentation steps has been introduced after each comma, exactly as **SQL formatter** would format this query.

```
1 MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
2 RETURN
3   p.name AS actorName,
4   p.birthdate AS actorBirthdate,
5   p.nationality AS actorNationality,
6   m.title AS movieTitle,
7   m.releaseYear AS movieReleaseYear,
8   m.genre AS movieGenre,
9   r.role AS rolePlayed
```

**Listing 10:** Cypher query with a long list of return items

# Chapter 3

## Methodology

---

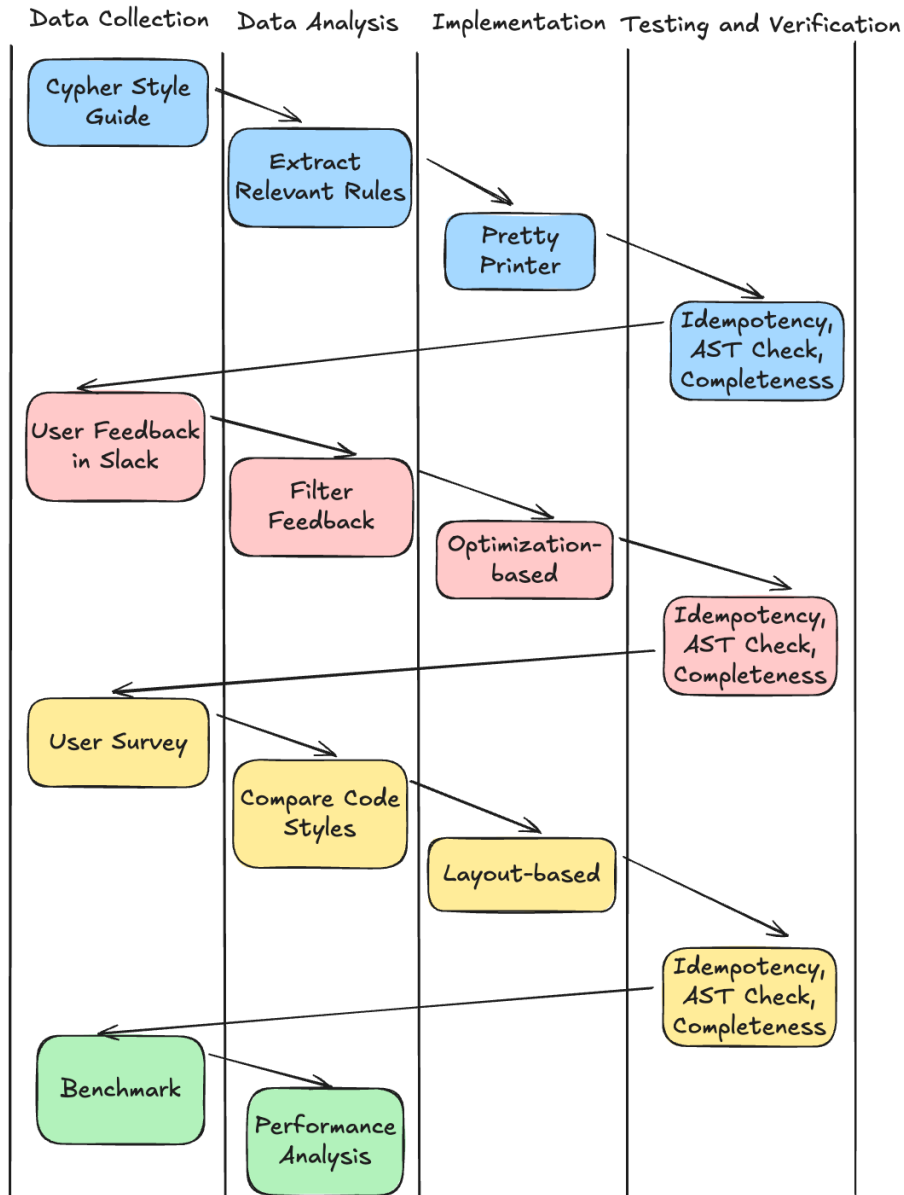
In this chapter, we outline the processes used to iteratively implement and evaluate the three formatting approaches outlined in Section 2.3. An overview of the process we followed is included in Figure 3.1. We cover the practical testing and verification methods that we used during development, as well as the methods we used to gather feedback to aid our decision-making during the entire process. The methodology is designed to address RQ2, and the insights from everything except the testing and verification phases forms the basis for addressing RQ1.

### 3.1 Data Collection

This section describes the type of data collected and the methods used, directly corresponding to the boxes in the first column of Figure 3.1. The first section describes our starting point for data collection, which was the Cypher style guide, as it served as a foundation for our formatter. Next, we discuss the data collected through user feedback in Slack and by a survey which supported development and helped us decide on a styling convention. Finally, we describe the benchmarking process carried out for each of the approach once they had all been implemented.

#### 3.1.1 Cypher Style Guide

The starting point of this thesis was to collect rules and recommendations from the Cypher style guide [30], because rather than creating divergent styling conventions, which might discourage usage, we instead wanted to complement the existing conventions. As a secondary source of information for styling conventions, we sometimes consulted the Cypher documentation when we felt that the style guide was ambiguous, or when it did not contain rules for a specific language construct.



**Figure 3.1:** The iterative process we followed to answer the research questions. Blue is first iteration, pink second, yellow third. Green is benchmark which was performed on all iterations.

### 3.1.2 User Feedback in Slack

To better understand the experience of our users with the formatter, we collected feedback through Neo4j’s internal slack channels. Upon publishing each iteration, we announced the release in relevant Slack channels. The announcement was intended to create a discussion and inform users that a new version was released and available for testing. These discussions with colleagues helped us generate interest around the formatter and to gain deeper insight into users’ opinions on the formatter. We received a total of twelve responses, with participants ranging in seniority from mid-level engineers to directors.

### 3.1.3 User Survey

Between versions two and three, we conducted a user survey to collect anonymous feedback on the preferred code style. The primary focus was to compare the output of the optimization-based and layout-based approaches, which is also why the survey was conducted between these versions. We present an excerpt of the the user survey in Figure 3.2 while the entire survey can be found in Appendix A.

\*

```

1 MATCH (p:Product)
2 RETURN p.name AS productName, p.price AS basePrices, p.price AS priceWithTax,
3        p.price AS basePrice, p.price AS priceWithTax
4 LIMIT 25;

```

12345

☆ ☆ ☆ ☆ ☆

\*

```

1 MATCH (p:Product)
2 RETURN
3   p.name AS productName,
4   p.price AS basePrices,
5   p.price AS priceWithTax,
6   p.price AS basePrice,
7   p.price AS priceWithTax
8 LIMIT 25;

```

12345

☆ ☆ ☆ ☆ ☆

Figure 3.2: An excerpt of the user study

We distributed the survey through three different Neo4j Slack channels. We chose these channels because it would give us a wide reception from individuals who both developed Cypher and used Cypher in their daily work. Two of the channels had slightly below 300 members, while the third was close to 400. This gave us a total of 20 participants for the survey.

The survey consisted of four different code snippets which we chose on the basis that they included Cypher code which we thought was representative of real-world queries, and because they could illustrate the contrast between the formatting approaches. For each snippet, the participants were presented with three to four ways of formatting it, resulting in a total of 13 questions. Each question asked the participant to rate the code snippet on a scale from one to five. All questions for a given snippet were presented on one page to ensure that the participants rated the code style relative to each other. The intention was to reveal whether a certain code style was preferred for just one code snippet or if it was a general preference.

### 3.1.4 Benchmark

To assess the performance of the three different approaches, we measured the time it took to format 10 000 queries from the sample query dataset described in Section 3.4.3. Because these queries are written by users of the Cypher language, we consider them representative of what real Cypher code looks like. Although we have access to more than a million real queries, we decided to limit the amount used to run the benchmark to just a small subset of the full dataset. This was done in part for practical reasons, as using the full dataset would take significantly longer (hours versus minutes), and we considered it unlikely to increase the quality of the measurement in a meaningful way. Using only the subset also made it easier for us to avoid background processes disturbing the execution, as the total runtime was much shorter.

We ran the benchmarks on a Macbook Pro, specifically a 2023 M3 Max model with 64 GB RAM, running macOS Sonoma 14.7.4. We selected commits 20a9625a [45], 0689b876 [44], and c86b44e4 [46] for each approach, respectively. The repository containing the formatter is open source [28], and the commits chosen for the pretty printer and the optimization-based formatter are simply the last before transitioning the formatter to the next approach.

Because the formatter is implemented using TypeScript and the benchmark ran for up to a minute, we did not expect warm-up effects to impact the results significantly. We used `ts-node v10.9.2` to execute the TypeScript code and ran the benchmark five times for each approach, measuring the execution time.

## 3.2 Data Analysis

In this section we describe how we analyzed the collected data, that is, activities related to the boxes in the second column of Figure 3.1. First, we discuss how the rules from the Cypher style guide were filtered to determine their relevance. Second, a discussion on how and why user feedback should be categorized. Third, an outline of how the user survey was analyzed. Finally, we discuss our method for assessing the performance of each approach.

### 3.2.1 Extract Relevant Rules

Initially, our intention was for our formatter to implement all the rules described in the style guide. However, we quickly realized that many of the recommendations, while very useful to developers using Cypher, do not necessarily fall under the responsibility of a formatter. We thus filter the rules on the basis of two criteria. First, because we consider a formatter to be a tool that primarily modifies whitespace, the rule should not change the underlying AST. Second, the rule should fall under the responsibility of a formatter rather than, for example, a linter. The rules were then classified as in- or out-of-scope on the basis of whether they met the criteria.

### 3.2.2 Filter User Feedback

An inherent challenge in evaluating a formatter is that the quality of the output is highly subjective. For this reason, we divide the feedback into two different categories: bugs and opinions. Of the total of twelve responses, three were classified as bugs, four as opinions, and the rest were seen as a combination of bugs and opinions. We categorized these according to whether the feedback included what we thought is something we clearly missed, then it was a bug. However, if the feedback was more of a preference, then it was categorized as an opinion.

The first category, bugs, was straightforward and easy to deal with; a bug is by definition bad and simply needs to be fixed. There were also a few cases where a user reported something as a bug when, in fact, it was the intended output of the formatter. This was mostly an issue in the first iteration of the formatter due to its simplicity.

The second category of user feedback was opinions about code style preferences. This category was more difficult to address because fulfilling a request by one person could be in conflict with the preferences of another. Our way of dealing with this problem was to try and weigh different opinions against the practices of other established formatters. If other formatters do something similar to what the feedback suggested, we added the feedback to our backlog and later implemented it. The opinions were added to the backlog and ranked according to how recently they were added and how difficult they were to implement. However, if the suggestion was more unconventional and we were unable to find any examples of other formatters doing the same, we had to make a judgement call as to whether other users would agree with the suggestion.

### 3.2.3 Compare Code Styles

To analyze the user survey, we compared the average rating for each question with the same style in different code pieces to find similarities. This allowed us to find out whether a certain code style was preferred for just one code snippet or if it was a general preference. Ideally, we should have statistically analyzed the results to determine if they were significant. However, we decided that statistical analyses were not necessary as the results were merely used as a guideline to be weighted with other factors.

### 3.2.4 Performance Analysis

From the five benchmark runs, we calculated the average run time along with the standard deviation. We then analyzed this in comparison to how fast we believe the formatter needs to be in order to be viable in scenarios such as a continuous integration and deployment (CI/CD) pipeline or a developer's workspace.

## 3.3 Implementation

As discussed in Section 2.3, the three approaches to formatting that we intended to examine share many similarities. In particular, if the maximum column width allowed is infinite, all three implementations should produce exactly the same output. Additionally, if dynamic line breaks are needed, both the optimization-based and the layout-based approach should group language constructs in the same way.

The similarities between implementations made us decide to implement each approach iteratively, as we could then reuse infrastructure between iterations. We began with implementing the pretty printer, which served as a baseline implementation, while subsequent approaches would tackle the problem of dynamic line breaks. After an initial exploration of the difficulty of implementing the optimization-based and layout-based approaches, the optimization-based approach appeared simpler and thus became the second iteration, while the layout-based approach became the last.

## 3.4 Testing and Verification

This section describes the testing and verification strategies that we used during the implementation of the three approaches. First, we present the testing carried out during development. Second, we specify which properties the formatter had to pass before releasing to users. Finally, we describe how the formatter was evaluated on a large sample to ensure correctness.

### 3.4.1 Testing Strategy

Testing of the formatter was limited to integration-level tests. Because the expected result was known, it was straightforward to create black-box tests that verified that the formatter output matched the expected result. Black-box testing was preferable over unit testing as it meant that we did not have to change a lot of tests when changing architecture. The only tests we changed between approaches were those related to line breaking.

### 3.4.2 Essential Formatter Properties

For a formatter, we concluded that there are three key properties that must hold to ensure correctness: idempotency, AST integrity, and non-whitespace character count. Violating any of these can lead to unpredictable behavior, making them essential checks. Our inspiration for the idempotency and AST integrity check came from Geirsson [16], while non-whitespace count was introduced based on our own observations during development.

**Idempotency**, the first integrity check, means that once a query is formatted, formatting it again should produce the exact same result. If this is not the case, a query may produce another new result with each formatting pass. While ensuring idempotency might seem straightforward, it can easily become quite complex, especially regarding the placement and preservation of comments.

**AST integrity**, the second integrity check, is used to verify that nothing disappears from the original text. This can, for example, happen if some parts of a visitor function are not fully implemented. Therefore, it is an important check that the AST remains identical before and after formatting. If the formatted output produces an AST that differs from the original, it could indicate that text has been lost or that a syntax error has been introduced, both of which are unacceptable.

**Non-whitespace character count**, the third integrity check, is used to verify that the number of non-whitespace characters does not change when formatting. This might at first glance seem identical to the AST check; however, because comments are not part of the AST, it is important to count all characters to make sure that no comments have been added or gone missing. Both problems occurred during implementation and were the reason this check was introduced. This is the only integrity check that runs directly with the formatter and not as a separate tool. The reason being that if something is added or missing, it indicates a serious error in the formatter, and it is better to throw an error than potentially return a faulty result.

### 3.4.3 Verification at Scale

In order to make sure that the formatter functioned as intended, it was essential that all the formatter properties, as described in Section 3.4.2, were validated on a large set of queries. This was done using a dataset of approximately one million queries. These queries were made by actual users, had been collected before this thesis, and were provided to us by a product analytics team. This large-scale sample played a crucial role in building confidence before releasing the formatter to real-world users. Due to the importance, we decided to include this in the CI/CD pipeline as well. This ensured that the integrity checks passed on approximately 15 000 queries before new features were merged into the main branch. The reasoning behind using only a subset of the entire dataset was that it had to be small enough to not bloat the GitHub repo. But this was only for smaller feature changes; before each new release, the entire dataset of a million queries had to pass the integrity checks. The complete validation process took approximately 30 minutes.

Another use case for the sample queries was that they enabled us to format previously unseen queries one by one and evaluate the result. This was very valuable in finding new ways to construct queries in the Cypher language. In addition to finding new constructions of the language, the dataset also enabled us to get a feel for how well the formatter was performing in terms of output quality.



# Chapter 4

## Formatting Rule Set for Cypher

---

In this chapter, we present our findings on how to define a comprehensive set of formatting rules for Cypher (RQ1). We first discuss which rules we consider to be in or out of scope. Following this, we present the results of our user survey and the feedback that we received. Based on those insights, we propose additions to the Cypher style guide that we believe are necessary for it to be unambiguous and comprehensive. The findings, as illustrated in Figure 4.1, complement the style guide and are based on the insights from our implementation, informal user feedback gathered during development, and the results of our survey.

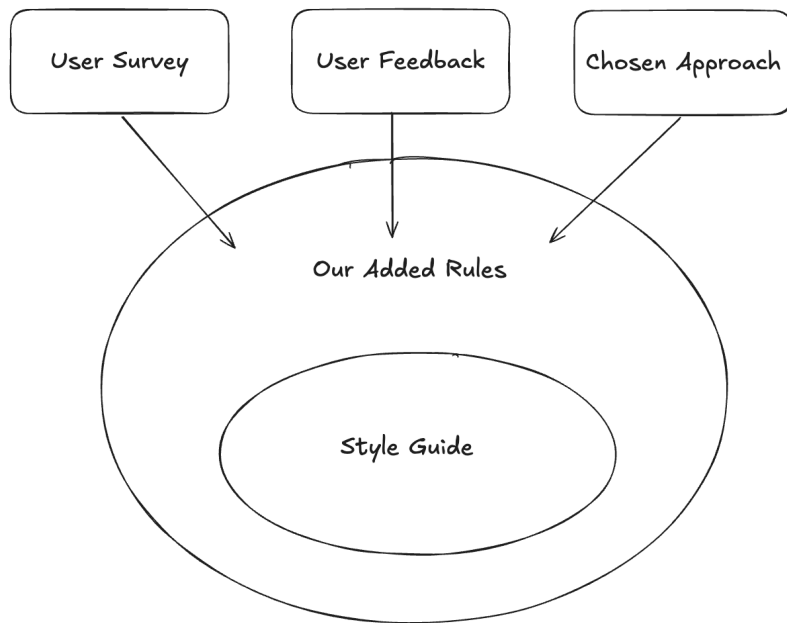
### 4.1 Scope of a Formatter

After filtering the rules of the style guide [31] according to our two criteria—that rules must not change the underlying AST and must fall under the responsibility of a formatter—we determined that seven out of 26 were out of scope. These rules were mostly related to the casing of variables and the ordering of constructs. One such rule, which we considered, but ruled out, was the recommendation for the ordering of `ON CREATE` and `ON MATCH`, described as follows:

*"Put ON CREATE before ON MATCH if both are present." (Cypher styleguide, rule 2)*

Reordering the `ON MATCH` and `ON CREATE` statements as shown in Listing 11 is recommended by the style guide since it is considered good practice. But since this transformation would alter the structure of the underlying AST—and, in our opinion, is more appropriately handled by a linter—we decided that it should fall outside of the scope of our formatter.

However, we did include two rules, despite them technically not meeting the criteria. One of those exceptions was to include the rule about converting keywords to uppercase. Similarly to SQL, idiomatic Cypher code uses uppercase for all keywords. This (arguably) makes it easier to read the queries, but it is quite inconvenient while typing it out. Upon



**Figure 4.1:** Illustration of how the user feedback, user survey and our chosen approach shaped our additions to the style guide

```
1 // BAD
2 MERGE (a:A)-[:T]->(b:B)
3   ON MATCH SET b.name = "you"
4   ON CREATE SET a.name = "me"
5 RETURN a.prop
6
7 // GOOD
8 MERGE (a:A)-[:T]->(b:B)
9   ON CREATE SET a.name = "me"
10  ON MATCH SET b.name = "you"
11 RETURN a.prop
```

**Listing 11:** ON MERGE / ON CREATE style guide example

examining the sample queries described in Section 3.4.3, we also noticed that many users strayed from this recommendation. Although we cannot confirm this, we suspect that the underlying reason was that it is simply inconvenient to type things in uppercase. Based on the fact that we considered the feature to provide a significant increase in terms of quality of life, we decided to include it, despite technically being out of scope since it changes the contents of the AST.

The other exception we made was occasionally changing the order of comments relative to how they appear in the source code. The reasoning was that the formatter should be allowed to make decisions without having to be constrained by the placement of comments. If that is not the case, a user might inadvertently make the formatter behave in odd ways through innocuously placed comments. To illustrate this principle, consider the code shown in Listing 12. If the formatter is not allowed to move comments, the **AND** keyword will be

```

1 // Formatting output if comments cannot be moved
2 WHERE
3   p.price > 100 // This comment forces the AND down
4   AND p.stock < 50
5
6 // Formatting output if the comment was not present
7 WHERE
8   p.price > 100 AND
9   p.stock < 50

```

**Listing 12:** Illustration of how comments may affect formatting if they cannot be moved. Note that the maximum column width is assumed to be reduced, making line breaks necessary. In the first **WHERE** clause the comment has forced the line break to occur before the **AND**, rather than after, as its grouping rule recommends.

forced onto the next line, breaking the grouping rule described in the rule for line breaks.

## 4.2 Survey Results and User Feedback

An overview of the results is presented in Table 4.1, where the questions have been divided into the types of formatting approach they represent. In particular, the layout-based approach consistently received the highest or second highest ratings across all of the examples included in the survey.

Unlike the user survey, the informal user feedback we received was generally not related to the underlying formatting approach we had chosen. Instead, it mostly centered on bugs, or idiomatic Cypher conventions that our formatter did not adhere to, due to us being unaware of them. We do not think this means that users found the formatting approach to be unimportant, but rather that it was easier to suggest concrete improvements to specific language constructs compared to suggesting another styling convention altogether. Several of the suggestions we present in Section 4.3 are based directly on the feedback we received

through these channels, which helped to ensure that our recommendations were relevant to actual practice.

| Question #                                 | Average Rating (1-5) |
|--------------------------------------------|----------------------|
| <b>Layout-based</b>                        |                      |
| 2                                          | 3.75                 |
| 6                                          | 3.80                 |
| 9                                          | 3.50                 |
| 10                                         | 3.50                 |
| 11                                         | 3.55                 |
| Category Average:                          | <b>3.62</b>          |
| <b>Layout-based with Extra Indentation</b> |                      |
| 3                                          | 2.60                 |
| 4                                          | 3.55                 |
| 7                                          | 3.20                 |
| 13                                         | 3.45                 |
| Category Average:                          | <b>3.20</b>          |
| <b>Optimization-based</b>                  |                      |
| 1                                          | 3.45                 |
| 5                                          | 2.75                 |
| 8                                          | 2.45                 |
| 12                                         | 2.35                 |
| Category Average:                          | <b>2.75</b>          |

**Table 4.1:** Average ratings by question on our styling survey, grouped by styling approach

## 4.3 Added Rules

This section presents the Cypher styling rules that we assess are missing from the style guide in its current state. Note that most language constructs in Cypher already have recommendations for how they should be styled available in the documentation. The part of Cypher that is by far the most lacking in terms of styling recommendations is line breaks, which is also the primary aspect that we have focused on improving.

### Rule: Maximum Column Width

*Limit line length to 80 characters if possible.*

Long lines have been shown to have a negative impact on code readability [35], and almost all industry standard formatters enforce a maximum column width. The exact limit generally varies between 80 and 100 characters [14, 49], and we found that 80 was, in our subjective opinions, the limit that resulted in the best output. The caveat "if possible" is included since

we want to acknowledge that exceeding the limit is sometimes unavoidable. If, for instance, an identifier is 81 characters long, there is not much one can do to avoid breaking the rule.

## Rule: Line Breaks and Indentation

*Line breaks should be applied starting from the outermost group, with each nested group breaking until either the line fits or no more line breaks can be inserted. Add two spaces of indentation for each nested broken group.*

The approach to line breaks and indentation that we decided was the best fit for Cypher was the layout-based approach. The basis of this was primarily that this type of styling received the highest rating for almost all questions in our styling survey. This is also the approach taken by the **SQL formatter** and the immensely popular **Prettier** formatter. These factors, combined with the fact that we personally prefer this style, give us confidence that this is the right approach for Cypher as well.

Note that we did not specify how constructs should be grouped in the definition of the rule. This is omitted mainly for brevity, because there is an immense amount of different groupings used in our implementation. These grouping rules also implicitly imply many styling rules. An example of this is that nodes and relationships are grouped together, which means that a pattern should always split before a node rather than a relationship if possible, as shown in Figure 4.2. To illustrate how grouping should be done, consider the following examples:

1. Clauses should be grouped together with the expression, pattern, or list that follows them.
2. Both nodes and relations should be contained within a group, and in longer patterns, a group should span one node and relationship at a time.
3. Binary expressions (for example, **AND**, **OR**, **+** and **-**) should make up one group, and, additionally, the operator and the left expression should become a group.
4. Lists should be grouped as a whole, and each list item should consist of its own group.

An example of the effect of following these recommendations for grouping and line breaks is shown in Figure 4.2.

## Rule: ORDER BY and LIMIT

*Treat ORDER BY and LIMIT as clauses in terms of styling.*

**ORDER BY** and **LIMIT** are technically not clauses despite looking very similar to, and generally being treated like, clauses. That means they are not covered by style guide rules such as "Start a new clause on a new line" [31]. Based on the way in which we observed them to be manually formatted by users and the informal feedback we received on them, we propose to treat them like clauses for formatting purposes. To illustrate the effect of applying this rule, consider the query shown in Listing 13.

```
1 MATCH (n:Person)
2 WHERE n.name = "Steve"
3 RETURN n
4 ORDER BY n ASC
5 LIMIT 12;
```

**Listing 13:** Example of **ORDER BY** and **LIMIT** being treated as clauses

## Rule: List and Map Literals

*Lines that end with a list or map literal and do not fit within the maximum width should leave the opening bracket/brace on the previous line and use only two additional spaces of indentation.*

A common pattern in **Prettier**, as shown in Listing 8, is to treat list and map literals differently if they are the only thing remaining on a line. This slightly reduces the amount of whitespace needed and was by far the most popular choice in our survey. We also considered some other less common alternatives, as illustrated in Listing 14. No other option was nearly as popular, and so, based on the fact that we want to emulate **Prettier** as much as possible, as well as its high rating on the survey, we propose that it should be part of the set of formatting rules for Cypher as well.

## Rule: CASE Expressions

*A group containing a CASE expression should always split, and each WHEN, ELSE, and END should be put on a new line. Additionally, WHEN and ELSE should add two spaces of indentation.*

The formulation of a styling rule for **CASE** expressions proved to be particularly challenging. Not only can they appear nearly anywhere in a query, but their clause-like grammar also hints that they should contain multiple forced newlines. While reviewing sample queries containing **CASE** expressions, we found that some users preferred to put the whole expression on one line, while others always split them up, treating them almost like clauses. An illustration of the different ways in which we observed users formatting **CASE** expressions can be seen in Listing 15. The option we deemed most reasonable, after examining a large number of queries containing this type of expression, was to always split it, as shown in Alternative 1.

## Rule: Explicit Newlines

*A single blank line may be used to separate clauses, queries or comments.*

One aspect of code style preference that quickly became obvious to us after analyzing user feedback is that users often insert blank lines within their code for clarity, and they do not appreciate those being removed. After releasing the first version of the formatter, one of the biggest points of feedback we got was that removing explicit newlines reduces clarity. We were initially of the opinion that the formatter should completely ignore the whitespace in

```

1 // Alternative 1 - Prettier-style (preferred)
2 RETURN [
3     "Item1",
4     ...,
5     "Item9"
6 ];
7 // Alternative 2 - Default layout-based
8 RETURN
9 [
10    "Item1",
11    ...,
12    "Item9"
13 ];
14 // Alternative 3 - Optimization-based
15 RETURN ["Item1", "Item2", "Item3",
16         "Item4", "Item5", "Item6",
17         "Item7", "Item8", "Item9"]

```

**Listing 14:** The different alternatives we considered for how to format lines that end with list/map literals and do not fit within the maximum width. Note that the maximum width is assumed to be shorter to illustrate how the list splits.

```

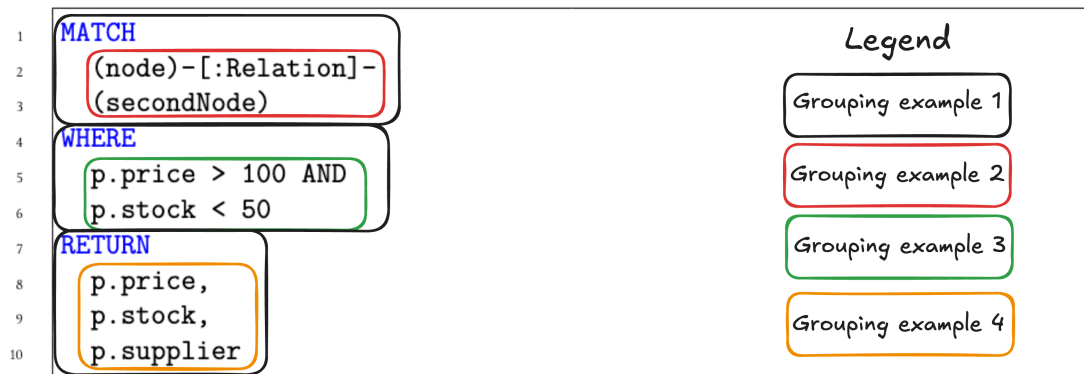
1 // Alternative 1 (preferred) - expressions, but always break
2 RETURN
3     CASE n.age
4         WHEN null THEN 1
5         ELSE n.age
6     END
7 // Alternative 2 - clause-like
8 RETURN
9 CASE n.age
10     WHEN null THEN 1
11     ELSE n.age
12 END
13 // Alternative 3 - inline
14 RETURN CASE n.age WHEN null THEN 1 ELSE n.age END

```

**Listing 15:** The different ways in which users had manually formatted `CASE` expressions in the sample queries we obtained.

the original code. But after examining how **Prettier** and other formatters handled this, it became clear that an exception should be made in this case.

However, we do not propose to allow blank lines everywhere. Only those that come before or after clauses, queries, or comments should be preserved. Our reasoning is that the formatter should have a high level of agency, and whitespace choices made by the user are secondary to that.



**Figure 4.2:** Illustration of the effect of applying the example grouping rules, where each colored box represents the region which it affects. Note that the second example rule implies that we should split before a node rather than a relation.



# Chapter 5

## Technical Design

---

In this chapter, we cover the technical implementation details of each version of the Cypher formatter, organized into three main sections, each covering one iteration. We discuss the data structures used, which were partly inspired by the concepts introduced in Section 2.2, and explore the complexity and performance characteristics of each version in detail. These insights form the foundation for our subsequent analysis of the trade-offs of each approach in Chapter 6.

### 5.1 Pretty Printer Approach

The pretty printer was the first approach we implemented, serving as the foundation for the other approaches. With this simple approach, we were able to implement all the rules of the style guide that we decided were within scope in just a matter of days. However, this approach did not have any support for dynamic line breaks and only broke lines when the style guide specifically required it, such as when a new clause started. Unfortunately, this lack of support for dynamic line breaks often led to the formatter reducing readability compared to manually formatted code. Nevertheless, it served as an important foundation for the upcoming iterations and allowed us to begin collecting valuable feedback from users.

Our pretty printer leveraged the ANTLR tooling described in Section 2.1 to generate a CST and a visitor class that could be used to systematically traverse it. Then, whitespace was manually inserted for the language constructs that needed it, such as newline characters for clauses and spaces between keywords. Indentation was handled in a similar way, where language constructs such as `CALL` and `CASE` would cause an increase in indentation by two steps after each newline for all of its children.

### 5.1.1 Formatting Example

In Listing 16, we present an example of a simple query before and after being formatted by our pretty printer. Note that each clause is put on a new line, the keywords are converted to upper case, and the spacing around the `>` operator is corrected.

```
1 // Before formatting
2 match (n) where n.prop>10 return n
3
4 // After formatting
5 MATCH (n)
6 WHERE n.prop > 10
7 RETURN n
```

**Listing 16:** Simple `MATCH-WHERE-RETURN` query formatted by the pretty printer.

## 5.2 Optimization-based Approach

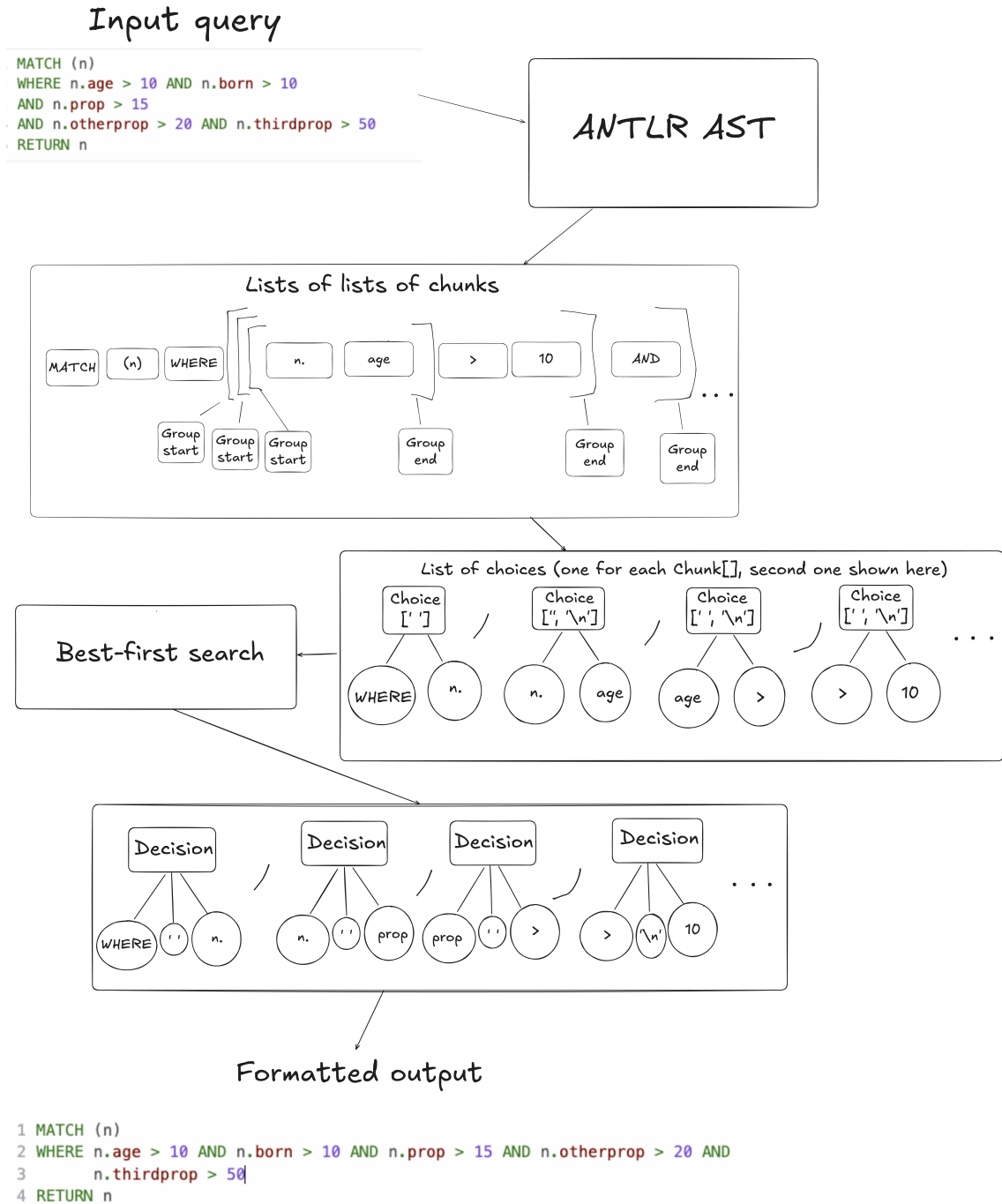
The implementation of the optimization-based formatter is, in our opinion, by far the most complex. It is especially complex at the code level, with a seemingly endless list of edge cases, but even conceptually it took us a while to fully grasp. Because of this, this second iteration also took us the longest to implement. It required roughly two whole months of development before we got it into production, at which point we still had to spend additional time resolving technical debt accrued along the way.

To make the implementation more digestible, we first present a high-level overview of how it works conceptually and the types used in our implementation. We also provide an architectural diagram in Figure 5.1 of all components that illustrates the flow from input to formatted query. Following the overview, we dive deeper into the implementation of the search function and the optimizations needed to make it run in a reasonable amount of time. Although much of the infrastructure from the previous iteration could be reused, some significant changes had to be made to enable the best-first search.

### 5.2.1 High-level Overview

Our optimization-based formatter builds on top of the pretty printer implementation, leveraging the visitor implemented previously. It retains all the simpler formatting choices, such as the style guide rule for spacing in maps, but also adds logic to construct instances of the new data structures introduced. The basic formatting (formatting that does not take line width into account) thus looks essentially the same in the second iteration as in the first, also supporting all the style guide rules that we decided were in scope.

Line breaks require an additional step, and the data structures are used to convert the formatting problem into a best-first search over a weighted graph to find the "best" solution.



**Figure 5.1:** Architectural overview of the second iteration of the Cypher formatter, which used the optimization-based approach.

In this case, the best solution is the one that breaks lines in the way that is most preferable, represented by a cost heuristic. Breaking a line increases the cost of a solution (which makes it less preferable), but exceeding the limit of 80 characters is always considered more costly than any number of line breaks. In order to efficiently ( $O(\log n)$  for each update) keep track of the best state at each step, a min-heap is used. The graph traversal stops as soon as a solution has been found, at which point it simply backtracks to the start of the query, keeping track of newlines and indentations along the way. Reconstructing this path then yields the resulting formatted output.

In this version of the formatter, the input query is divided between each part that has a forced line break, such as each clause. The reason for doing this is to reduce the size of the search space, although it does mean that the formatter cannot reason about formatting choices that should span across those forced line breaks.

## 5.2.2 Data Structures

This subsection covers the data structures used by the visitor that enable the best-first solution search. Their actual interfaces are included for reference, but note that some unimportant implementation details may have been omitted for brevity or changed for clarity. Of course, there are far more types than these in the actual implementation, and the ones that are not required to understand the architecture have been omitted. The majority of data structures that we ended up using in the final implementation of the optimization-based formatter are derived from concepts found in `scalafmt` [16], as we found those to be the most applicable.

### Chunk

```
1 interface Chunk {  
2   chunkType: 'REGULAR' | 'SYNTAX_ERROR' | 'COMMENT';  
3   text: string;  
4   indentation: number;  
5   groupsStarting: Group[];  
6   groupsEnding: Group[];  
7   spacingOptions: 'noSpace' | 'noBreak' | 'mustBreak';  
8 }
```

Listing 17: TypeScript interface for the `chunk` type

Our `chunk` type is very similar to the one described in Section 2.2. It represents a string that always stays together and contains information on indentation, spacing choices, and whether it is grouped together with other `chunks`. As Nystrom puts it [34]:

*"Chunks are similar to a token in a conventional compiler, but they tend to be, well, chunkier."*

As will become apparent in Section 5.2.3, the size of the search space scales with the number of **chunks** we have. We therefore want to make them as large as possible, as this will significantly impact the performance of the formatter. To illustrate how tokens are "chunked" together, consider the pattern in Listing 18. In our implementation, the pattern is converted

```

1 MATCH (NODE01)-[REL01]->(NODE02)
2
3 // Tokens
4 [MATCH, (, NODE01, ), -, [, REL01, ], -, >, (, NODE02, )]
5 // Chunks
6 [MATCH, (NODE01, )-, [REL01, ]->, (NODE02, )]
```

**Listing 18:** Simple Cypher pattern used to illustrate "chunking"

into **6** separate **chunks**, while originally consisting of **13** distinct tokens, providing a considerable performance gain. In addition to any performance gains, "chunking" tokens together reduces the likelihood of introducing formatting errors, such as a space being inserted between **NODE01** and **)**, for instance.

## Group

```

1 interface Group {
2   align: number;
3   breakCost: number; // Calculated as 10^(nesting level)
4 }
```

**Listing 19:** TypeScript interface for the **group** type

Our concept of a **group** is partly inspired by the **groups** described by Wadler [50], and partly by the concept of "spans" in **dartfmt** [34]. Similarly to how a **chunk** groups tokens together, a **group** groups **chunks** together. A **group** can also group other **groups**, and if visualized as a tree, its children become either **chunks** or other **groups**. However, a key difference between **chunks** and **groups** is that **groups** allow whitespace to be inserted between its children. Generally, the formatter can choose between inserting a newline or a space between the children of a **group**, and inserting newlines is penalized according to the nesting level of that **group**. The primary function of a **group** is thus to inform the solution search about which **chunks** are related. This enables indentation of related constructs and helps the search understand which solutions are preferable.

To illustrate how **chunks** are grouped, consider the query shown in Listing 20. Here, the whole expression in the **WHERE** clause is grouped together, to indicate that we would prefer it to stick together without any line breaks. Furthermore, the parts that make up the OR expression are also grouped together, so splitting after **=** would be penalized even more.

```
1 WHERE a = 1 OR b = 1
2
3 // Groups visualized, where [ indicates the start of a group, and ]
  indicates the end.
4 WHERE [[a = 1] OR] [b = 1]]
```

**Listing 20:** Example of formatting groups

## Split

```
1 interface Split {
2   splitType: ' ' | '' | '\n' | '\n\n';
3   cost: number;
4 }
```

**Listing 21:** TypeScript interface for the `split` type

A `split`, inspired by the concept of the same name in `scalafmt` [16], represents the whitespace between two `chunks`. The most common types of `split` are `' '` and `'\n'`, although an empty string or two newlines is sometimes also an option. The double-newline option was used in cases where a user had explicitly inserted an empty line between two different clauses, queries, or adjacent to a comment. In those cases, our formatter, much like `Prettier`, will preserve that newline since it was likely put there intentionally to increase readability. We preserve at most one empty line; if there are two or more, they are reduced to a single blank line.

In addition to the type of spacing choice that a `split` represents, it also contains a cost. This cost is generally `-1` for an option that does not break the line (to incentivize putting as much as possible on one line before breaking), and `1` otherwise. Note that if the spacing choice is contained within a `group`, the cost of breaking the line becomes the nesting level of the `group` to the power of 10 instead. This ensures that line breaks within nested groups are always more costly than breaks at lower nesting levels. The choice of  $10^n$  for the cost of breaking nested groups is somewhat arbitrary and was chosen to ensure that breaking nested groups is always more costly. Using, say,  $12^n$  instead would likely work just as well; the important part is that nested line breaks are heavily disincentivized.

## Choice

---

```

1 interface Choice {
2   left: Chunk;
3   possibleSplitChoices: Split[];
4   right: Chunk;
5 }

```

**Listing 22:** TypeScript interface for the `choice` type

A `choice` encapsulates the possible ways in which the gap between two `chunks` can be filled. This type is similar to, and inspired by, the `FormatToken` type in `scalafmt` [16]. Generally, the possible `split` choices are a space and a newline, but in some cases the `choice` is limited to only one option. The possible `split` choices in a `choice` eventually become the weighted edges in the graph over which we perform a best-first search. Note that the last element in the list of `choices` has an empty `chunk` as its right child, which is not included in the final result.

## Decision

```

1 interface Decision {
2   indentation: number;
3   left: Chunk;
4   right: Chunk;
5   chosenSplit: Split;
6 }

```

**Listing 23:** TypeScript interface for the `decision` type

The `decisions` are the objects used to reconstruct the formatted string based on the result of the best-first search; to reconstruct it, we simply add the indentation, the text of the left `chunk`, and the chosen `split` to our result for each decision taken.

## State

The `state` type represents a node in the weighted graph described in Section 5.2.1 and is inspired by the `state` types in `dartfmt` [34] and `scalafmt` [16]. It keeps track of the `chunks` between which it is currently choosing a `split` and the cumulative cost of the `state`. A `state` represents a candidate solution in the best-first search, and thus the heap queue that sorts solutions has the type `Heap<State>`.

### 5.2.3 Best-first Solution Search

The solution search works very similarly to how a Dijkstra search over a graph works; we put all of our nodes (`states` in this case) into a heap, explore them in the "best-first" order, and

```
1 interface State {  
2   activeGroups: Group[];  
3   column: number;  
4   choiceIndex: number;  
5   cost: number;  
6   overflowingCount: number;  
7 }
```

**Listing 24:** TypeScript interface for the `state` type

stop as soon as we reach our goal. The `states` are sorted such that we primarily compare them on the number of characters that exceeded the 80-character limit (`overflowingCount` in the state type), and if that number is equal, we compare their cost. Note that we could not make the 80-character limit a hard limit, as there are cases where all possible solutions exceed it at some point. Using a best-first search allows us to visit far fewer states than with a breadth-first search, for example, and also enables us to stop as soon as we find a solution; if we find a solution, it has to be the best one, since we always look at the best state first.

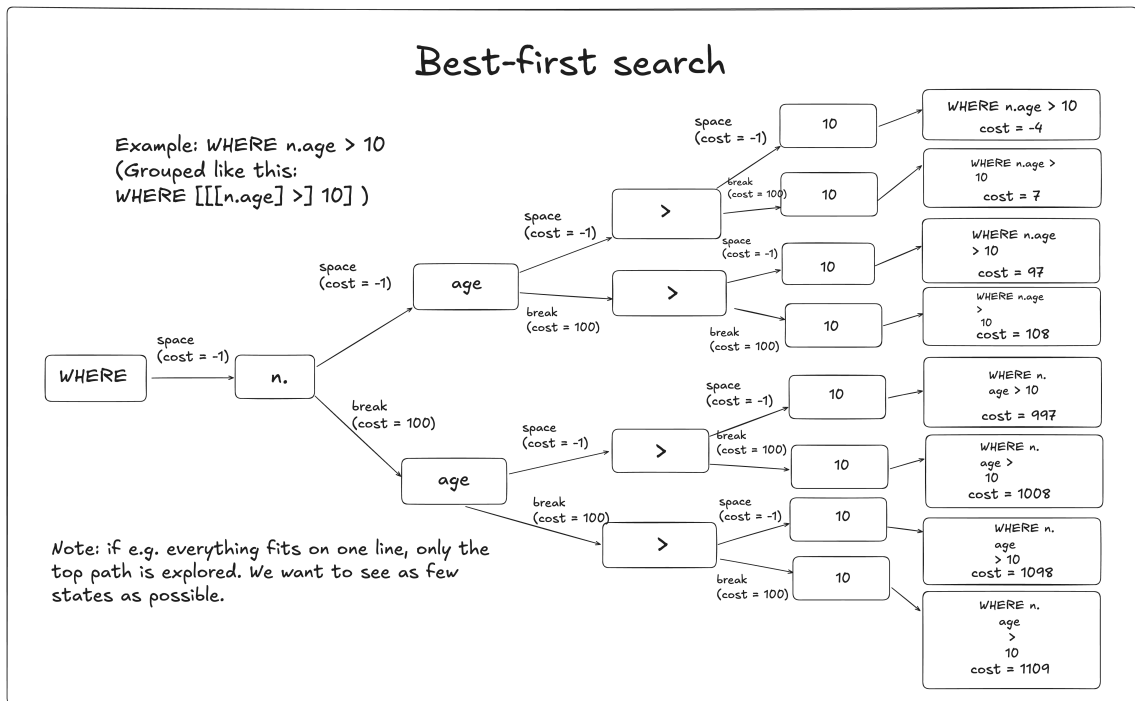
Although the best-first search allowed us to skip many states, the underlying problem of choosing between two `split` types between each pair of `chunks` is still exponential. Consider, for example, the search space when formatting a simple `WHERE` clause, as illustrated in Figure 5.2. Despite the clause consisting of only five different chunks, we still ended up with eight different potential solutions. Assuming that each choice has two `splits` to choose from, the theoretical time complexity for the search is  $O(2^n)$ , where  $n$  represents the number of choices we had to make (roughly equal to the number of `chunks`).

## 5.2.4 Dynamic Programming

A reader familiar with dynamic programming might instinctively reach for some form of caching in this case, but will then face a surprisingly difficult challenge: How do you choose a state/cache key? A naive approach would be to simply make the state key the index of the current choice being made. This would result in a very attractive time complexity of  $O(n)$ , but comes with a major issue: It is completely incorrect. Any line break also affects how subsequent line breaks should be made, and so this key does not encapsulate all the information required to represent a state.

Consider the code in Listing 25. Both states 1 and 2 get the exact same state key, meaning that state 2 would never be explored if caching on only the choice index. A natural solution to this might be to instead use the history of all previous choices as the state. They could then be represented as a string of bits, 0 indicating a non-breaking `split` choice, and 1 indicating a breaking one. But now we are back to a total of  $O(2^n)$  possible states, and memoization has not provided us with any performance benefits. What we aim to illustrate with this short example is the trade-off with respect to granularity in the state key; a finer granularity reduces the risk of us missing out on the best solution while worsening the time complexity.

Although simple forms of caching like the ones described above are not applicable, optimization is vital to get a formatter that ran in a reasonable amount of time. If you were to



**Figure 5.2:** Example of a search space in the best-first search part of the optimization-based formatter

```

1  // Code to be formatted
2  WHERE a = 1 OR b = 1
3
4  // State 1, next chunk: '=', choice index: 4
5  WHERE a = 1 OR
6      b
7
8  // State 2, next chunk: '=', choice index: 4
9  WHERE a = 1 OR b

```

**Listing 25:** Illustration of why an index is not sufficient for the state key

format a query consisting of 75 **chunks** (not at all an unreasonable amount), and assuming an average of 10 ns spent visiting each state, it would have taken up to **11979620 years** to find a solution in the worst case. Other formatters, such as **dartfmt** or **scalafmt** have gotten around this using various ad hoc optimizations, such as aggressively pruning the search space or providing hints in the form of "optimal tokens" to the search that help guide it toward finding the best solution as fast as possible. They also both include an "escape hatch" in the case that they do not find a solution within some predefined large number of searched states. This is necessary as their searches are still  $O(2^n)$  in the worst case, though in practice they found that the escape hatch is triggered almost exclusively in pathological cases [34, 16]. These approaches are much more complex than simply introducing dynamic programming, so we opted to start by experimenting with cache key granularity. The goal was to find a middle ground where we find the optimal/intended solution in as many cases as possible, without having to spend an exorbitant amount of time writing ad hoc optimizations.

In our first attempt at dynamic programming, we used (**choiceIndex**, **column**) as our state key, where **choiceIndex** is the index of the **choice** currently being considered, and **column** indicates the solution's position in the current line. Our reasoning was that it is unlikely that two different solutions will end up in the same column if their splits are not the same, thus reducing the risk of false positives. This brought the theoretical time complexity from  $O(2^n)$  to  $O(\text{MAX\_COL} \cdot n)$ , which is, of course, a massive improvement. Our formatter would now always find a solution in linear (albeit with a large factor) time, though it might still miss some optimal solutions. Situations such as the one in Listing 25 would no longer be an issue; the first state would have the key (4, 7), and the second (4, 16). This worked relatively well and was, in fact, the state key used in the initial launch of the optimization-based formatter.

However, although this worked in most cases, there were still a good number of queries where the search failed due to this key choice. Consider the two different versions of the same query in Listing 26. Clearly, the second version should be preferable, since it avoids one additional line break within the long list. But with just (**choiceIndex**, **column**) as the cache key, this solution seemingly disappeared from the solution search due to a false cache key hit. Although this was a rare occurrence, it happened often enough that we started assuming that any new bugs were caused by false cache hits. Not being able to trust the accuracy of the caching thus not only gave our users worse solutions, it also made the predictability of the formatter far worse, significantly slowing down development. In short, we needed a more granular cache key, or some other form of optimization that was more reliable.

After much experimentation, we modified the state key to instead consist of (**choiceIndex**, **column**, **lastSplit**), where **lastSplit** represents the column at which the most recent line break was made. Some other attempts included using the current line number or the last few split choices, but none of these provided anything close to the accuracy provided by the **lastSplit** field. Our solution search would now find the intended solution in all of our test cases, and there was no indication that any useful states were being skipped. This, of course, does not mean that the caching is now objectively correct; it is almost certainly possible, at least in theory, to find scenarios where false cache hits occur. But in practice, which we consider to be the only worthwhile scenario to consider, it works flawlessly. We have yet to find a counterexample in which an incorrect solution is produced because of it.

Given that a line break can occur in any column, this addition to the key increases the theoretical time complexity to  $O(\text{MAX\_COL}^2 \cdot n)$ . An 80-fold increase if we assume a max-

```
1 // Solution found using [choiceIndex, column]
2 MATCH (p:Product)
3 WHERE p.price > 1000 AND p.stock > 50 AND
4       p.category IN ["Electronics", "Home Appliances",
5                      "Sports Equipment", "Automotive Parts",
6                      "Fashion Accessories", "Books", "Toys",
7                      "Jewelry", "Musical Instruments",
8                      "Art Supplies", "Office Supplies"]
9 RETURN p
10
11 // Intended solution
12 MATCH (p:Product)
13 WHERE p.price > 1000 AND p.stock > 50 AND
14       p.category IN
15       ["Electronics", "Home Appliances", "Sports Equipment",
16        "Automotive Parts", "Fashion Accessories", "Books",
17        "Toys", "Jewelry", "Musical Instruments", "Art Supplies",
18        "Office Supplies"]
19 RETURN p
```

**Listing 26:** Illustration of why an index together with the current column is not sufficient for the state key

imum column width of 80. In practice, however, after benchmarking the solution against 10000 real-world queries, we found that the average time to find a solution increased only by about 35%. While this is of course a substantial drop in terms of performance, it was a small price to pay in comparison to having an unreliable caching mechanism that would frequently yield incorrect solutions.

### 5.2.5 Alignment and Indentation

The optimization-based version of our formatter uses a combination of indentation and alignment operators to vertically align related constructs for readability. The indentation works much like the one implemented in the first iteration, while the alignment is a mechanism within the best-first search. In short, it means that if a **group** splits, any **chunk** beginning on a new line will be vertically aligned with the column at which the **group** initially split. Note that alignment takes precedence over indentation and any indentation will be ignored if a split occurs within a **group**. To illustrate this concept, consider the code shown in Listing 26. The `p.category IN` part of the long **AND** expression lines up on the same column as the `p.price` part. Similarly, each time the list breaks, the new elements are also put on the same column as where the first item of the list started.

At first glance, this might look like it would lead to all code being crammed into the rightmost part of the 80-column width limit. But one should remember that line breaks are heavily penalized, and line breaks within (nested) language constructs even more so. In practice, this means that the solution search generally prefers solutions that break as early as possible to avoid having to split multiple times at a high alignment value.

Alas, to every rule there are exceptions. In the case of Cypher, the exceptions are the **CASE** expression and its siblings. As its name suggests, it is an expression and can thus appear nearly anywhere within a query. It also requires indentation and mandatory newlines between its children. This forced us to add another concept of "aligned indentation", since the **CASE** expression should not only line up after, for example, the clause keyword, but also indent its children in addition to the alignment. This enabled output such as the one shown in Listing 27.

```
1 RETURN CASE
2     WHEN purchaseCount > 0 THEN "Active"
3     ELSE "Inactive"
4 END AS status
```

**Listing 27:** A (relatively simple) CASE statement formatted by the optimization-based version of the formatter

The **CASE** expression is also not the only construct that requires specialized indentation. There are also **COUNT**, **EXISTS**, and **CALL** expressions, each with different rules. Additionally, the fact that some of them can contain nested clauses means that we have to format them as effectively separate queries. Although we were able to produce reasonable results for all of these types of expressions, we ended up with complex logic to handle the different types of

indentation. By the time we finished implementing the optimization-based approach, we had three different types of indentation, and the technical debt slowed the development process significantly.

## 5.2.6 Formatting Example

In Listing 28, we present an example of the output of our optimization-based formatter. Note how the algorithm tries to fit as much as possible onto one line, and when a chunk does not fit, it is placed on a new line and aligned with the start of the **group**.

```

1 // Before formatting
2 match (n:Node)-[R:relation]->(p:Person)
3 return ["itemOfList1", "itemOfList2", "itemOfList3", "itemOfList4",
4         "itemOfList5"]
5
6 // After formatting
7 MATCH (n:Node)-[r:Relation]->
8   (p:Person)
9 RETURN ["itemOfList1", "itemOfList2",
10         "itemOfList3", "itemOfList4",
11         "itemOfList5"]

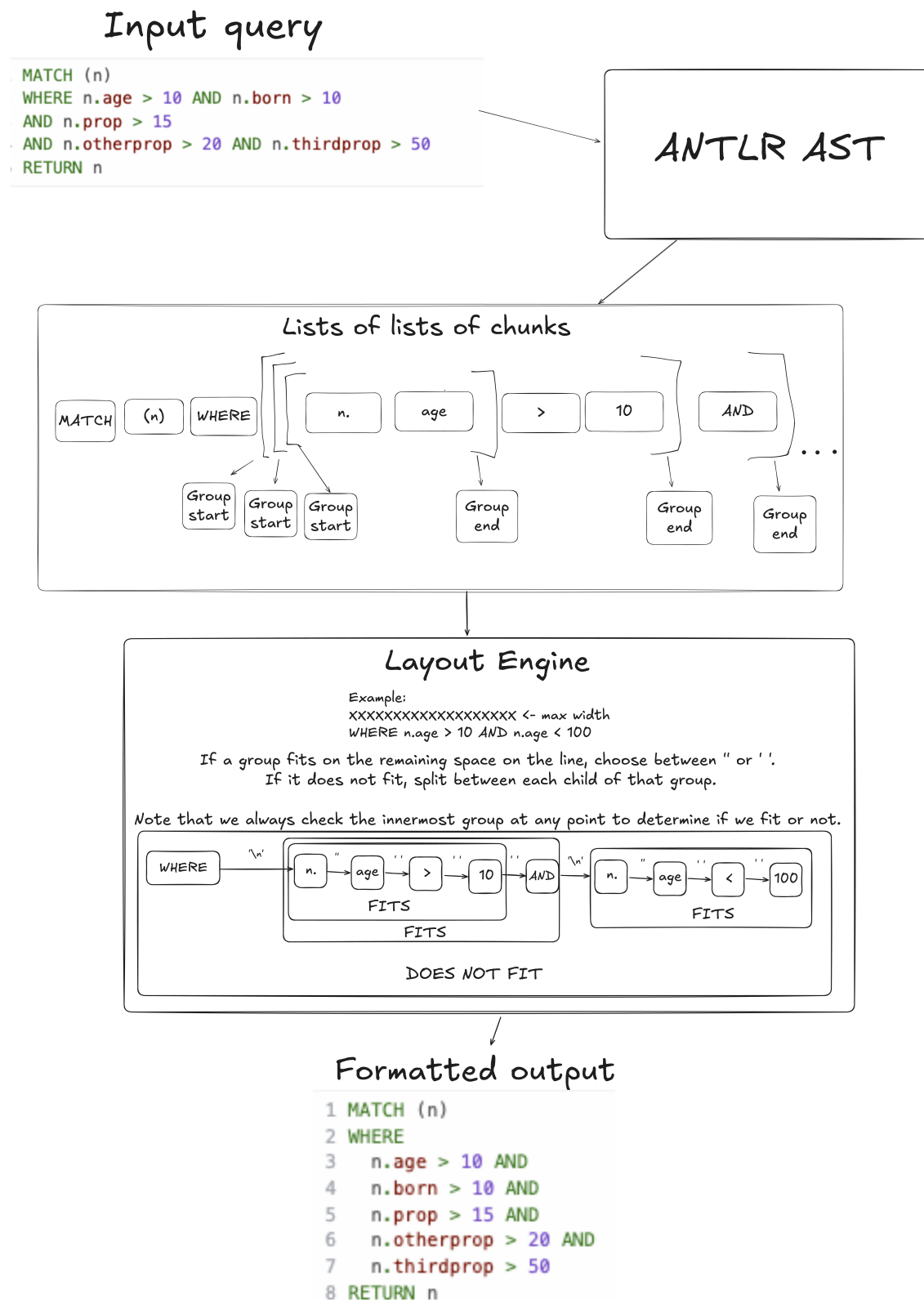
```

**Listing 28:** A simple MATCH-RETURN query, shown before and after formatting with our optimization-based formatter.

## 5.3 Layout-based Approach

Our implementation of the layout-based approach builds on the implementations of the previous approaches, leveraging the pretty printer, as well as the grouping logic introduced with the optimization-based approach. However, compared to the optimization-based approach, its line-breaking algorithm and indentation logic differ significantly. These differences also led to a much simpler architecture, with fewer steps from input to output, and a more streamlined design. An overview of the new architecture can be seen in Figure 5.3.

We replaced the entire solution search with what we call a layout engine, as illustrated by the central box in Figure 5.3. The layout engine is used to apply the layout algorithm outlined in the **Prettier** paper by Wadler [50], inserting either spaces or newlines between each pair of **chunks** depending on whether or not the **groups** need to split. Removing the solution search also meant that we could remove data types such as **split**, **choice**, and **decision**, reducing the complexity. Because the layout engine always comes to the same conclusion as to whether or not a **group** should split, there is no longer a choice to be made for each whitespace between chunks. This means that it can process all the **chunks** linearly, reducing the theoretical time complexity from  $O(\text{MAX\_COL}^2 \cdot n)$  to  $O(n)$ . Removing the solution search also meant that we no longer had to split the query into separate lists of **chunks** to



**Figure 5.3:** Architectural overview of our implementation of the layout-based approach. Note the similarities and differences to the optimization-based approach.

reduce the search space. This led to improved output quality in places where we previously could not reason about line breaks between lists of chunks.

The layout engine also uses much simpler logic for indentation compared to the previous iteration. In the optimization-based approach, because of the reliance on group alignment, we required three different types of indentation. Transitioning away from this type of alignment allowed us to reduce the number of indentation types to just one.

### 5.3.1 Determining Group Split

The primary function of the layout engine is to, for each **group**, determine whether the **group** needs to be split or not. We identified two main approaches to this. Our solution is based on the idea that each **group** has a **size**, which is the total number of characters that it consists of, including spaces. The **group** sizes are precomputed before the list of **chunks** enters the layout engine, enabling it to immediately determine whether or not a group fits on the current line when it encounters it. If the sum of the **group** size and the current column exceeds the maximum column width, the **group** should split between all of its children.

The other option, which appears to be more popular based on our review of the literature, is to use some form of look-ahead. This approach first attempts to format the **group** without splits, but if this exceeds the maximum column width, the group is split. A caveat to this is that a group which chose not to split might have to do so later on, in order to stay within the maximum column width. This causes all the sub-**groups** to that group to also re-evaluate whether they need to put their **chunks** on new lines.

It is not immediately clear which option is better. Our method requires pre-computing the **group** sizes; however, it enables linear processing which makes debugging a lot easier. In contrast, look-ahead does not require any pre-computation and might offer a performance benefit in some cases.

### 5.3.2 Separation of Indentation and Groups

A mistake we made that greatly complicated the implementation was not separating the logic of grouping and indentation. Examining how **Prettier** works, it might seem reasonable to deduce that when a **group** splits, the indentation level increases. In fact, this is often what happens in practice. Therefore, one would think that the indentation is controlled by a **group**. That is, when a **group** is split, the indentation level also increases, making **group** the controller of the indentation level.

This perspective on **groups** and indentation resulted in us spending considerable time exploring various suboptimal solutions, none of which worked as intended. One attempt included different types of **group** where some could and some could not change the level of indentation. Another attempt reduced the number of **groups**, in an attempt to avoid excessive indentation. However, both attempts led to excessive or insufficient indentation, resulting in incorrect formatting.

The solution was to completely separate the indentation logic from the **group** logic. This solved the issue where the incorrect amount of groups would cause improper indentation, and an issue we had of having to choose which group should apply indentation when only one of them should. Instead of worrying that adding a group might mess up indentation, we could group related terms without having to think about indentation, and we could similarly

modify indentation as we saw fit. For example, when a list is encountered, a **group** is added around all items, and the indentation level increases before and decreases after the list. This has the result that if the **group** fits on one line, the indentation is never applied because no split is needed. But if the **group** splits, each item will go on a new line and will have added indentation.

### 5.3.3 Layout Engine Variants

Compared to how Wadler [50] describes how to implement the layout-based approach, we have done something slightly different. His implementation constructs a tree structure called a document, and this hierarchy of documents is then traversed much like an AST. However, even though we wanted similar results, we did not take the same path of constructing a tree. The main reason was that, because of our iterative approach to development, we already had the list structure from the optimization-based approach. This led us to implement the layout-based approach with a list instead of a tree.

The advantage of using a list structure is the simplicity of implementing and debugging it. A list is very easy to debug, as you can step through each item linearly, inspecting the program's state at each step. Compared to a tree which needs to have a state to keep track of, making the tree comparatively harder to understand, especially when debugging.

Based on how the layout-based algorithm is described in Wadler's paper [50] and the implementations of it that we found online, using a list similar to what we did in our implementation is quite rare. We have, in fact, not been able to find any implementation of a layout-based formatter that uses a list rather than a tree, despite it being the easier approach to grasp in our experience. Unless there are other implementations that we are not aware of that also use a list rather than a tree-like structure, our implementation thus also provides a unique interpretation of the layout-based algorithm.

### 5.3.4 Formatting Example

Listing 29 shows an example of Cypher formatted by our layout-based formatter. Note that the **MATCH** clause has split and indentation has been added at the start of the pattern. The pattern itself has also split, resulting in a newline after the first node-relationship pair. The list is split into individual lines for each item, with the opening bracket preserved on the first line in accordance with the formatting rule for list literals.

```
1 // Before formatting
2 match (n:Node)-[R:relation]->(p:Person)
3 return ["itemOfList1", "itemOfList2", "itemOfList3", "itemOfList4",
4         "itemOfList5"]
5
6 // After formatting
7 MATCH
8     (n:Node)-[r:Relation]->
9     (p:Person)
10  RETURN [
11      "itemOfList1",
12      "itemOfList2",
13      "itemOfList3",
14      ...
15  ]
```

**Listing 29:** Simple MATCH-RETURN query before and after formatting using the layout-based approach



# Chapter 6

## Evaluation of Approaches

---

In this chapter, we document our findings related to output quality, performance, and maintainability for each of the three approaches to implementing a formatter (RQ2). We rank the dimensions according to how critical we find them to be for a formatter and weigh our evaluation of the approaches accordingly.

We consider output quality to be by far the most important metric, under the caveat that performance should not be completely unreasonable. Simply put, the vast majority of developers who interact with the formatter will care only about its output, not its performance, nor how maintainable it is. We would consider execution time exceeding 500 ms for an average-sized query to be unreasonable for any output quality, since this would significantly impact responsiveness and usability, and likely render the formatter impractical for many CI/CD workflows. The exact threshold for what counts as reasonable can obviously vary, but 500 ms is roughly what we consider a "perceptible delay" for formatting, and such delays may prove irritating to the user [13].

After output quality, we consider performance to be the next most important metric. Formatters are generally expected to be very performant, to the point that you can run them as part of either a pre-commit hook, a CI/CD pipeline, or both. In those cases, the formatter might need to format a whole repository of files every time a new commit is made to it. At that point, the performance of the formatter runs the risk of being a bottleneck, and optimizing it becomes critical.

Maintainability, although a very important metric for any piece of software, is what we consider to be the least important among these three. The programming languages being formatted rarely change drastically over time, with only perhaps a couple of new additions per year. In addition, the requirements for formatting rarely change as styling preferences are generally rigid over time. However, maintainability is certainly not unimportant. Using a more complex approach might lead to more errors during implementation, as well as an increase in the amount of developer time needed to implement and maintain the formatter.

## 6.1 Output Quality

As noted in Section 2.3, all three approaches produce the same formatting output in the case that no line exceeds the maximum column width. Additionally, as discussed in Section 4.3, we consider dynamic line breaks to be a critical feature for a formatter. As a result, we will automatically consider the first iteration, the pretty printer, to have the worst output quality. The optimization-based and layout-based formatters will thus be evaluated purely on the style of line breaks they allow.

As hinted at by the fact that the rules we propose for line breaks in Section 4.3 directly correspond to the rules of a layout-based formatter, we strongly prefer its output. The optimization-based formatter enables far more complex layouts, but in the case of Cypher, this does not seem to come at any benefit. The output of the optimization-based version of the formatter was included in the styling survey and, as presented in Table 4.1, consistently received the worst scores of all available alternatives. We believe this is due to the fact that Cypher has relatively simple styling conventions and can thus get away with a simpler approach. Nystrom, who used the optimization-based approach for `dartfmt`, mentions that Dart users prefer to write many callbacks and long method chains [34]—a type of convention that is difficult to format, but is not used in Cypher. Because Cypher does not follow these types of convention, the layout-based approach is not only much simpler but also produces better output. In short, the limitations of the layout-based approach are not a weakness, they are a strength.

To summarize, we consider the pretty printer to have by far the worst output quality since it does not support dynamic line breaks, and the layout-based style to be better than the optimization-based style for Cypher specifically.

## 6.2 Performance

The theoretical time complexities for the pretty-printing approach, the optimization-based approach and the layout-based approach were  $O(n)$ ,  $O(\text{MAX\_COL}^2 \cdot n)$ , and  $O(n)$ , respectively. We thus expect the optimization-based approach to be the slowest, and the pretty printing approach to be slightly faster than the layout-based approach, since it stops ahead of the layout step. The benchmark results, as shown in Table 6.1, confirm this hypothesis. Although the time complexity of the optimization-based approach is in theory 6400 times slower than the others, in practice the difference is not as extreme.

**Table 6.1:** Benchmark results per approach.

| Approach           | Mean time | Standard deviation | Mean time/query |
|--------------------|-----------|--------------------|-----------------|
| Pretty Printing    | 33.34 s   | 0.35 s             | 3.3 ms          |
| Optimization-based | 54.54 s   | 0.78 s             | 5.4 ms          |
| Layout-based       | 38.92 s   | 0.42 s             | 3.9 ms          |

The optimization-based approach is the only one that stands out in terms of performance, with an average formatting time roughly one and a half times that of the others. However, we do not consider this to be a significant difference; someone using the formatter in their

editor is very unlikely to be able to discern a difference of 2 ms. Even though it might play a bigger role if used in, for example, a CI/CD pipeline, we believe that any user would gladly take the performance hit if it meant the formatter produced better output.

## 6.3 Maintainability

Although the underlying complexity of a formatter is generally much less important to a user than its output quality and performance, it is highly important to the developer who implements and maintains it. In order to give a representative overview of how hard it is to implement and maintain each approach, we will evaluate the complexity of each approach and compare them to each other. The evaluations are in large part based on the complexities of each of our implementations, as described in Chapter 5.

In terms of complexity, as the pretty printing approach does not include any dynamic line breaking, it becomes the simplest to develop and maintain. The comparison is then between the optimization-based approach and the layout-based approach. In our view, the layout-based approach is the clear winner. This comes down to the difference in complexity of how dynamic line-breaking works. As stated in Section 5.3, the optimization-based approach relies on a heap, best-first search, and a wider range of types and concepts. In contrast, the layout-based approach encapsulates all logic within a `for` loop, significantly reducing complexity.

Although lines of code (LOC) does not necessarily determine complexity, it may offer an indication; focusing only on the algorithm for dynamic line breaking, the optimization-based approach had approximately 450 LOC while the layout-based approach was just shy of 200 LOC.

Comparing how indentation works between the optimization-based and layout-based approach also shows why the latter is easier to maintain. As stated in Section 5.3, the transition from optimization-based to layout-based reduced the complexity of the indentation state by lowering the number of different indentation values needed from three to one. The need for three different indentation levels often made maintaining and adding new features quite complex, as everything else had to play nicely with these indentation levels.

By eliminating the need to reduce the search space, as stated in Section 5.3, not only did the quality of the output improve, but maintainability also increased. The complexities of handling the transfer between states were not straightforward. The need to transfer some state variables between search spaces was error prone and hard to maintain. Another benefit of removing the search space and replacing with a `for` loop is that debugging becomes easier. Debugging the best-first search is challenging, primarily because the only information available at each step is the current state. This is further complicated by the fact that the state needs to be as minimal as possible to conserve memory. In contrast, it is easier to debug the layout-based approach, which operates linearly.



# Chapter 7

## Limitations

---

We present clear findings regarding both our evaluation of approaches and Cypher styling preferences, but to properly contextualize them, certain limitations must be acknowledged. In this chapter, we discuss these limitations, specifically addressing issues related to the validity and generalizability of our results.

### 7.1 Styling Preferences

For RQ1, the primary threat to the validity of our findings is whether or not the rules we propose are representative of actual styling preferences among Cypher users. In terms of user feedback, the users of the language we consulted could pose an external threat to validity as they might not be representative of the entire population of users. Our respondents consisted mainly of Neo4j employees and, in particular, engineers who develop the language itself. The styling preferences of those engineers might not align that well with users who use the language for something like data science, as that poses entirely different challenges for readability. One of the ways we tried to counteract this was by exposing the formatter and the survey to users from an internal Slack channel called `#field-team`. This channel is used mainly by engineers in customer-facing roles, meaning they spend the majority of their time solving actual problems faced by real users using Cypher. Since this is the type of user for which we were trying to optimize readability, we expected it to make the results more representative. Nevertheless, we suspect that some of the feedback we gathered and drew conclusions from came from users who might use the language in unconventional ways. Furthermore, many respondents to the survey, as well as many of the users from whom we received informal feedback, were aware that we had both implemented the formatter and proposed the styling rules. It is possible that this inadvertently led them to provide more positive feedback out of politeness, as is common when asking for feedback on one's own work [11].

Another factor that could impact both the external and the internal validity of our findings is which languages we draw inspiration from. Although we tried to look at styling con-

ventions for query languages such as SQL as much as possible, our styling choices ultimately closely resemble that of general-purpose code. While this kind of styling does seem to be very popular also in query languages, we are not necessarily convinced that the same principles always apply. On a similar note, it is possible that some of the styling choices we recommend might not extrapolate well to general-purpose languages either.

## 7.2 Implementations of Approaches

The greatest internal threat to RQ2, as we see it, is how we applied the approaches laid out in Section 2.3 in practice. We took a lot of liberties in how we defined our data structures and designed our algorithms, occasionally to the point where the implementations no longer resembled the designs from which we drew inspiration. This is especially noticeable in the optimization-based approach, where we used a mixture of ideas from multiple existing formatters. It is possible that we inadvertently made the implementation more complex and inefficient than it needed to be in doing so, which could skew the results in favor of the layout-based approach unfairly. However, at the same time, one could argue that the increased number of implementation errors is simply a result of the underlying design being more complex. Our impression is that we made the design slightly more complex than it needed to be, but ultimately the complexity arose from the fundamental properties of the approach.

The iterative approach could also impact the findings in a meaningful way. For one, it might be a lot easier to implement a formatter the second or third time, regardless of the approach, since you naturally gain experience along the way. In our case, we additionally leveraged working code from previous iterations to implement the next ones. This could make the maintainability or complexity of an approach seem better or worse, depending on the quality of the previous iteration.

As noted multiple times throughout this thesis, the fact that Cypher is a query language is an important distinction to make. It is especially important in terms of external threats to validity for the implementation, as findings for a query language may not apply for general-purpose ones. Consider, for instance, the ways `dartfmt` supports splitting functions, as shown in Listing 7. Dart code often chains long method calls together [34], and we suspect that this is the reason the formatter wants to support unique styles. Cypher code is generally simpler in terms of syntax, and the layout-based style suits the majority of language constructs very well. This might skew the results unfairly in favor of the layout-based approach in terms of general-purpose languages; if the target language for the formatter has complicated styling conventions among its users, the findings do not necessarily apply. At the same time, the formatter that we took the most inspiration from was **Prettier**, which formats JavaScript. JavaScript is a general-purpose language and has some unique conventions, and the layout-based approach still seems to work remarkably well.

# Chapter 8

## Conclusions

---

To answer RQ1, Section 4.3 proposes six new additions to the Cypher style guide, with a plethora of rules also implied by the rule for line breaks. These rules, combined with the existing rules provided by the Cypher style guide, provide what we believe to be a comprehensive set of rules for Cypher that address existing ambiguities. By addressing the issues of ambiguity and incompleteness and enforcing the full set of rules through automatic formatting, we hope to see greater consistency in terms of styling for all codebases using Cypher.

Our second research focus, as described in RQ2, was the evaluation of different formatting approaches. We have identified the **layout-based** approach as by far the best for formatting Cypher, and our results indicate it could be preferable for other languages as well. Despite its simplicity and, thus, high maintainability, it provided the output that received the highest ratings in our survey on code styling. It also provided performance comparable to that of a very simple pretty printer, without needing complex performance optimizations.

We believe that these findings provide valuable insight for anyone aiming to implement a new formatter, for a query language, or otherwise. By addressing the lack of comparable research on what formatting approaches exist—and the benefits of each—this thesis provides a blueprint for formatter design that can greatly aid future development of formatters.

In terms of practical outcomes, the formatter we describe in this thesis has been released in production in multiple environments and is on track to become the official Cypher formatter. At the time of writing, it has been used by over 850 unique users via the Neo4j query tool, and we estimate that roughly as many users have used it in environments not tracked by usage metrics. It is also included in the official Neo4j VS-Code extension[26], which currently has more than 12600 downloads.

## 8.1 Future Work

Our findings strongly point to the layout-based approach being preferable in almost all aspects. However, as noted in Chapter 7, Cypher is a relatively simple query language, and we

cannot confirm that these results would be applicable for a general-purpose one. We believe that an evaluation of the three different approaches for a general-purpose language could provide further insight into which approach is better in that scenario.

Another aspect we think would be interesting to look into is whether or not it is possible to augment the layout-based approach such that it can also produce output similar to that of the optimization-based approach. This could help address the limitations of the layout-based approach with regard to more complicated layouts. Some research has already been done in this area [39], but we think including a comparison to the optimization-based approach could be of interest.

# References

---

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc., 2nd edition, 2006.
- [2] Prettier Authors. Prettier. <https://www.npmjs.com/package/prettier>, 2025. Accessed: 24-Apr-2025.
- [3] Prettier Authors. Prettier: An opinionated code formatter. <https://prettier.io/>, 2025. Accessed: 04-May-2025.
- [4] Prettier Authors. Prettier configuration docs. <https://prettier.io/docs/configuration>, 2025. Accessed: 19-Mar-2025.
- [5] Prettier Authors. Prettier technical details. <https://prettier.io/docs/technical-details.html>, 2025. Accessed: 19-Mar-2025.
- [6] The Go Authors. gofmt: Source code of the go formatter. <https://github.com/golang/go/blob/master/src/cmd/gofmt/gofmt.go>, 2025. Accessed: 11-Apr-2025.
- [7] R. Baecker. Enhancing program readability and comprehensibility with tools for program visualization. In *Proceedings. [1989] 11th International Conference on Software Engineering*, pages 356–366, 1988.
- [8] Raymond P.L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, July/August 2010.
- [9] Community. sql-formatter. <https://www.npmjs.com/package/sql-formatter>, 2025. Accessed: 24-Apr-2025. The package boasts hundreds of thousands of weekly downloads, indicating that it is one of the most popular tools for formatting SQL queries.
- [10] DB-Engines. Db-engines ranking. <https://db-engines.com/en/ranking>, 2025. Accessed: 18-Feb-2025.

- [11] Nicola Dell, Vidya Vaidyanathan, Indrani Medhi, Edward Cutrell, and William Thies. "yours is better!": participant response bias in hci. CHI '12, page 1321–1330, New York, NY, USA, 2012. Association for Computing Machinery.
- [12] Django Software Foundation. pyproject.toml configuration file. <https://github.com/django/django/blob/main/pyproject.toml>, 2025. Includes configuration for Black under `[tool.black]`.
- [13] David W. Embley and George Nagy. Behavioral aspects of text editors. *ACM Comput. Surv.*, 13(1):33–70, March 1981.
- [14] Free Software Foundation. Gnu coding standards. <https://www.gnu.org/prep/standards/standards.pdf>, n.d. Accessed 24-Apr-2025.
- [15] Python Software Foundation. Black: The uncompromising python code formatter. <https://github.com/psf/black>, 2025. Accessed: 19-Mar-2025.
- [16] Ólafur Páll Geirsson. scalafmt: opinionated code formatter for scala. Master's thesis, École polytechnique fédérale de Lausanne, Lausanne, Switzerland, June 2016.
- [17] Go Team at Google. Go for Visual Studio Code. <https://marketplace.visualstudio.com/items?itemName=golang.Go>, 2025. Accessed: 05-May-2025.
- [18] Hideo Hattori. autopep8. <https://pypi.org/project/autopep8/>, 2025. Accessed: 05-May-2025.
- [19] John Hughes. The design of a pretty-printing library. In *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1*, pages 53–96. Springer, 1995.
- [20] inferrinizzard. Prettier sql vscode. <https://marketplace.visualstudio.com/items?itemName=inferrinizzard.prettier-sql-vscode>, 2023. Accessed: 30-Apr-2025.
- [21] ISO/IEC. Iso/iec 9075:2023. <https://www.iso.org/standard/76583.html>, 2023. Accessed: 05-Apr-2025.
- [22] Stephen C. Johnson. Yacc: Yet another compiler compiler. <https://www.cs.utexas.edu/~novak/yaccpaper.htm>, 1975. Accessed: 24-Apr-2025.
- [23] Tab Atkins Jr., Erika J. Etemad, Florian Rivoal, Chris Lilley, and Sebastian Zartner. CSS Snapshot 2024. <https://www.w3.org/TR/2025/NOTE-css-2024-20250225/>, 2025. W3C Group Note, 25 February 2025.
- [24] Thomas Kanoutas, Thomas Karanikiotis, and Andreas L. Symeonidis. Enhancing code readability through automated consistent formatting. *Electronics*, 13(11), 2024.
- [25] LLVM Project. Jasper slides. Presentation slides, LLVM Dev Meeting, April 2013. <https://llvm.org/devmtg/2013-04/jasper-slides.pdf>, 2013. Accessed: 11-Apr-2025.

- 
- [26] Neo4j. Neo4j for vs code. <https://marketplace.visualstudio.com/items?itemName=neo4j-extensions.neo4j-for-vscode>, 2025. Accessed: 19-May-2025.
- [27] Inc. Neo4j. Neo4j graph database: The most trusted database for intelligent applications. <https://go.neo4j.com/rs/710-RRC-335/images/Neo4j-product-brief-database-US-EN.pdf>, 2022. Accessed: 18-Feb-2025.
- [28] Inc. Neo4j. Cypher language support. <https://github.com/neo4j/cypher-language-support>, 2025. Accessed: 31-Mar-2025.
- [29] Inc. Neo4j. Cypher manual: Introduction. <https://neo4j.com/docs/cypher-manual/current/introduction/>, 2025. Accessed: 31-Mar-2025.
- [30] Inc. Neo4j. Cypher manual: Overview. <https://neo4j.com/docs/cypher-manual/current/introduction/cypher-overview/>, 2025. Accessed: 31-Mar-2025.
- [31] Inc. Neo4j. Cypher styleguide. <https://neo4j.com/docs/cypher-manual/current/styleguide/>, 2025. Accessed: 18-Feb-2025.
- [32] Inc. Neo4j. graphgems.cypher. <https://github.com/neo4j/neo4j/blob/703ab467f4e75ef102f4900173926d716d3ab232/community/cypher-shell/cypher-shell/src/test/resources/org/neo4j/shell/parser/graphgems.cypher>, 2025. Git commit 703ab467f4e75ef102f4900173926d716d3ab232. Accessed: 31-Mar-2025.
- [33] Inc. Neo4j. How much faster is a graph database, really? <https://neo4j.com/news/how-much-faster-is-a-graph-database-really/>, 2025. Accessed: 24-Apr-2025.
- [34] Bob Nystrom. The hardest program i've ever written. <https://journal.stuffwithstuff.com/2015/09/08/the-hardest-program-ive-ever-written/>, 2015. Accessed: 5-Mar-2025.
- [35] Delano Oliveira, Reydney Santos, Fernanda Madeiral, Hidehiko Masuhara, and Fernando Castor. A systematic literature review on the impact of formatting elements on code legibility. *Journal of Systems and Software*, 203:111728, 2023.
- [36] openCypher Project. opencypher: The open source project for cypher query language. <https://opencypher.org/>, 2025. Accessed: 31-Mar-2025.
- [37] D.C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(4):465–483 – 483, 1980.
- [38] Terence Parr. ANTLR (ANother Tool for Language Recognition). <https://www.antlr.org/>, 2024. Version 4.13.2.
- [39] Sorawee Porncharoenwase, Justin Pombrio, and Emina Torlak. A pretty expressive printer. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):Article 261, 2023.
-

- [40] LLVM Project. clang-format: A tool to format C/C++/Java/JavaScript/Objective-C/Protobuf code. <https://clang.llvm.org/docs/ClangFormat.html>, 2025. Accessed: 2025-05-04.
- [41] pytest-dev. pyproject.toml configuration file. <https://github.com/pytest-dev/pytest/blob/main/pyproject.toml>, 2025. Includes configuration for Black under [tool.black] or related tooling.
- [42] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: A case study at google. In *International Conference on Software Engineering, Software Engineering in Practice track (ICSE SEIP)*, 2018.
- [43] sql-formatter Contributors. sql-formatter: A whitespace formatter for different sql dialects. <https://github.com/sql-formatter-org/sql-formatter>, 2025. Accessed: 11-Apr-2025.
- [44] Cypher Language Support. Commit 0689b876. <https://github.com/neo4j/cypher-language-support/commit/0689b876>, 2025. Commit on 10-Apr-2025.
- [45] Cypher Language Support. Commit 20a9625a. <https://github.com/neo4j/cypher-language-support/commit/20a9625a>, 2025. Commit on 07-Mar-2025.
- [46] Cypher Language Support. Commit c86b44e4. <https://github.com/neo4j/cypher-language-support/commit/c86b44e4>, 2025. Commit on 15-Apr-2025.
- [47] T. Tenny. Program readability: procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9):1271 – 1279, 1988.
- [48] The Go Authors. Effective Go. [https://go.dev/doc/effective\\_go](https://go.dev/doc/effective_go), 2022. Accessed: 05-May-2025.
- [49] Guido Van Rossum, Barry Warsaw, and Nick Coghlan. Pep 8 – style guide for python code. <https://peps.python.org/pep-0008/>, 2001. Accessed: 24-Apr-2025. PEP 8 recommends a maximum line length of 79 characters, reflecting the long-standing 80-column convention.
- [50] Philip Wadler. A prettier printer. *The Fun of Programming, Cornerstones of Computing*, pages 223–243, 2003.

# Appendices



# Appendix A

## User Survey

---

## Cypher Style Preference

This form will display various Cypher queries along with possible variations of outputs. Please rate each example in relation to others with identical code.

\* Indicates required question

\*

```
1 MATCH (b)
2 WHERE b.id IN ["8tArXrXtB", "ktSmaWFL", "7L2o6tDo", "1NVMr97k", "5NDcL7S9",
3              "FNMx1ztw", "W0cCK4i4", "1deBy9b6", "J14V40BF", "E1H0ndgH",
4              "VaCz6hZH", "Dg5u2vio", "2FL4hPr", "01uGk08U"]
5 MATCH path1 = (g)
6 RETURN b
7
```

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| ☆ | ☆ | ☆ | ☆ | ☆ |

\*

```
1 MATCH (b)
2 WHERE b.id IN [
3   "8tArXrXtB",
4   "ktSmaWFL",
5   "7L2o6tDo",
6   "1NVMr97k",
7   "5NDcL7S9",
8   "FNMx1ztw",
9   "W0cCK4i4",
10  "1deBy9b6",
11  "J14V40BF",
12  "E1H0ndgH",
13  "VaCz6hZH",
14  "Dg5u2vio",
15  "2FL4hPr",
16  "01uGk08U"
17 ]
18 MATCH path1 = (g)
19 RETURN b
```

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| ☆ | ☆ | ☆ | ☆ | ☆ |

```
1 MATCH (b)
2 WHERE b.id IN [
3   |"8tArXrXtB",
4   "ktSmaWFL",
5   "7L2o6tDo",
6   "1NVMr97k",
7   "5NDcL7S9",
8   "FNMx1ztw",
9   "W0cCK4i4",
10  "1deBy9b6",
11  "J14V40BF",
12  "E1H0ndgH",
13  "VaCz6hZH",
14  "Dg5u2vio",
15  "2FL4hPr",
16  "01uGk08U"
17 ]
18 MATCH path1 = (g)
19 RETURN b
```

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| ☆ | ☆ | ☆ | ☆ | ☆ |

#### Return Clause

\*

```
1 MATCH (p:Product)
2 RETURN p.name AS productName,
3        p.price AS basePrices,
4        p.price AS priceWithTax,
5        p.price AS basePrice,
6        p.price AS priceWithTax
7 LIMIT 25;
```

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| ☆ | ☆ | ☆ | ☆ | ☆ |

\*

```
1 MATCH (p:Product)
2 RETURN p.name AS productName, p.price AS basePrices, p.price AS priceWithTax,
3        p.price AS basePrice, p.price AS priceWithTax
4 LIMIT 25;
```

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| ☆ | ☆ | ☆ | ☆ | ☆ |

\*

```
1 MATCH (p:Product)
2 RETURN
3   p.name AS productName,
4   p.price AS basePrices,
5   p.price AS priceWithTax,
6   p.price AS basePrice,
7   p.price AS priceWithTax
8 LIMIT 25;
```

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| ☆ | ☆ | ☆ | ☆ | ☆ |

## OR expressions

\*

```

1 MATCH (p:Person)
2 WHERE p.name = 'Alberta' OR
3       p.name = 'Berta' OR
4       p.name = 'C' OR
5       p.age > 30 OR
6       p.salary > 50000 OR
7       p.experience > 10 OR
8       p.position = 'Manager'
9 RETURN p

```

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| ☆ | ☆ | ☆ | ☆ | ☆ |

\*

```

1 MATCH (p:Person)
2 WHERE p.name = 'Alberta' OR p.name = 'Berta' OR p.name = 'C' OR p.age > 30 OR
3       p.salary > 50000 OR p.experience > 10 OR p.position = 'Manager'
4 RETURN p

```

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| ☆ | ☆ | ☆ | ☆ | ☆ |

\*

```

1 MATCH (p:Person)
2 WHERE p.name = 'Alberta'
3 OR p.name = 'Berta'
4 OR p.name = 'C'
5 OR p.age > 30
6 OR p.salary > 50000
7 OR p.experience > 10
8 OR p.position = 'Manager'
9 RETURN p

```

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| ☆ | ☆ | ☆ | ☆ | ☆ |

```

1 MATCH (p:Person)
2 WHERE
3   p.name = 'Alberta' OR
4   p.name = 'Berta' OR
5   p.name = 'C' OR
6   p.age > 30 OR
7   p.salary > 50000 OR
8   p.experience > 10 OR
9   p.position = 'Manager'
10 RETURN p

```

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| ☆ | ☆ | ☆ | ☆ | ☆ |

### Map expression

\*

```
1 MATCH (p:Product)
2 WHERE p.inStock = true
3 WITH p, {
4   departmentId: 'DEPT-' + p.category,
5   taxRate: 0.08,
6   tags: p.tags,
7   vendorCode: p.vendorId,
8   dateAdded: p.createdAt
9 } AS productDetails
10 RETURN p.name
11 LIMIT 25;
```

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| ☆ | ☆ | ☆ | ☆ | ☆ |

\*

```
1 MATCH (p:Product)
2 WHERE p.inStock = true
3 WITH p, {departmentId: 'DEPT-' + p.category, taxRate: 0.08, tags: p.tags,
4   vendorCode: p.vendorId, dateAdded: p.createdAt} AS productDetails
5 RETURN p.name
6 LIMIT 25;
```

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| ☆ | ☆ | ☆ | ☆ | ☆ |

\*

```
1 MATCH (p:Product)
2 WHERE p.inStock = true
3 WITH
4   p,
5   {
6     departmentId: 'DEPT-' + p.category,
7     taxRate: 0.08,
8     tags: p.tags,
9     vendorCode: p.vendorId,
10    dateAdded: p.createdAt
11   } AS productDetails
12 RETURN p.name
13 LIMIT 25;
```

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| ☆ | ☆ | ☆ | ☆ | ☆ |



# Appendix B

## Generative AI Prompts

---

ChatGPT 4.5 ▾

I want better words for wasting and yielding part:  
This perspective on \mintinline{text}{group}s and indentation resulted in us wasting a lot of time exploring various suboptimal solutions which yielded nothing.

Here's a refined version with improved wording:

This perspective on \mintinline{text}{group}s and indentation led us to expend considerable effort investigating various suboptimal solutions, none of which produced meaningful results.

Or slightly more concise:

This approach to \mintinline{text}{group}s and indentation caused us to spend significant time evaluating various ineffective solutions that failed to yield productive outcomes.

Either choice enhances readability and maintains an academic tone.

Could i write something "none of which worked as intended"?

Yes, that's also a clear, concise, and appropriate phrasing. Here's how that would look:

This perspective on \mintinline{text}{group}s and indentation caused us to spend considerable time exploring various suboptimal solutions, none of which worked as intended.

This version directly communicates your point effectively and maintains a professional tone suitable for academic writing.

ChatGPT ▾

Convert to if statement:  
let finalIndent =  
state.activeGroups.length > 0  
? state.activeGroups.at(-1).align  
: state.indentationState.special;

Resonerat om if-sats konvertering i 4 sekunder >

```
javascript Kopiera
let finalIndent;
if (state.activeGroups.length > 0) {
  finalIndent = state.activeGroups.at(-1).align;
} else {
  finalIndent = state.indentationState.special;
}
```

No else, just one if statement

```
javascript Kopiera
let finalIndent = state.indentationState.special;
if (state.activeGroups.length > 0) {
  finalIndent = state.activeGroups.at(-1).align;
}
```



**MASTER THESIS** An Evaluation of Approaches to Code Formatting

A Case Study on the Cypher Query Language

**STUDENTS** Tomas Nyberg, Simon Thuresson**SUPERVISORS** Emma Söderberg (LTH), Oskar Damkjær (Neo4j)**EXAMINER** Görel Hedin (LTH)

# How to write a code formatter

POPULAR SCIENCE SYNOPSIS **Tomas Nyberg, Simon Thuresson**

A code formatter is an essential tool for every developer, which makes it all the more frustrating when a language like Cypher, the language used with the Neo4j database, does not have one. In our thesis, we implement a formatter for Cypher and evaluate the best approach to do so.

A code formatter is a tool that makes code more readable by fixing its white space, as shown in the example. Many developers consider it a must-have tool in a their tool belt. So when such a tool is missing, a demand for implementing one arises. But implementing a formatter is not as straightforward as you might think.

```
1 // Before
2 match (n) where n.age >25 return
   n.name
3 // After
4 MATCH (n)
5 WHERE n.age > 25
6 RETURN n.name
```

Our thesis evaluates three different approaches to implementing a formatter for Cypher, which is, curiously, missing a formatter, despite being used by the world's most popular graph database. In addition to providing guidelines for how to implement a formatter, we provide a complete formatter for Cypher available open source. Moreover, we complement Cypher's currently ambiguous style guide with additional formatting rules.

The three approaches were pretty printing, optimization-based, and layout-based formatting. Each approach was assessed by output quality, performance, and maintainability. To support our

development and enhancement of the style guide, we conducted a survey on Cypher styling preferences, and gathered user feedback.

After implementing all three approaches, we concluded that the pretty printer approach is by far the easiest to implement, and its performance is great. However, the output quality is very poor, as it does not try to keep the line width within a certain limit, which is crucial for readability. Because of this, the competition of being the best approach goes down to layout-based and optimization-based. The layout-based approach wins over the optimization-based approach in all three areas we evaluate.

We justify the win, first, by the fact that the user feedback and user survey showed that the layout-based had the best output quality. Second, it relies on a much simpler algorithm, which makes it faster. Third, we also found the layout-based approach to be much easier to implement and maintain, partly because it has fewer lines of code, roughly 200 versus 450. As a result of layout-based approach win over both optimization-based and pretty printing, we conclude that it is by far the best approach for Cypher, and likely also for other languages.