Graph Databases for Fast Queries in UD Treebanks

Niklas Deworetzki[1] **and Peter Ljunglöf**[1,2]
[1]Department of Computer Science and Engineering,
Chalmers University of Technology and University of Gothenburg
[2]Språkbanken Text, University of Gothenburg
nikdew@chalmers.se, peter.ljunglof@gu.se

Abstract

We investigate if labelled property graphs, and graph databases, can be a useful and efficient way of encoding UD treebanks, to facilitate searching for complex syntactic phenomena.

We give two alternative encodings of UD treebanks into the off-the-shelf graph database Neo4j, and show how to translate syntactic queries into the graph query language Cypher.

Our evaluation shows that graph databases can improve query times by several orders of magnitude, compared to existing approaches.

1 Motivation

Universal Dependencies (UD; de Marneffe et al., 2021) has celebrated 10 years of existence and has become a mature framework for text annotation. Currently there are almost 300 UD treebanks for almost 170 languages (Zeman et al, 2024).

One prominent use case of UD treebanks is to find examples of syntactic phenomena, within or across languages. E.g., Weissweiler et al. (2024) investigated if it is possible to identify grammatical constructions in different languages by searching for morphosyntactic patterns. Some of the queries they came up with were quite complex – they needed to cover all possible tree structures for a given construction, and at the same time rule out alternative interpretations.

There are several tools for searching in syntactic treebanks, such as ANNIS3 (Krause and Zeldes, 2014), AlpinoGraph (Kleiweg and van Noord, 2020), PML Tree Query (Štěpánek and Pajas, 2010), and Grew-match (Guillaume, 2021; Bonfante et al., 2018). They support complex queries and are usually efficient enough to be used on the existing UD treebanks. For example, Grew-match returns results within a few seconds, even when running a complex query on the largest UD treebank (Borges Völker et al., 2019).

If we are only interested in performing searches in manually annotated treebanks, the current tools are probably good enough. However, there are plenty of *automatically* annotated very large corpora.[1] If we want to perform searches for complex syntactic phenomena in such large treebanks (10–100 million tokens or more), the current query tools are not efficient enough. So there is a need for alternative approaches.

In this paper we investigate if existing off-the-shelf graph databases can be useful and efficient as a backend for complex searches in treebanks. We do this by giving two possible ways of encoding UD trees as *labelled property graphs*, which is the format underlying the Neo4j graph database (Francis et al., 2018). We also show how to translate Grew-match queries into the graph query language Cypher, and perform an extensive evaluation of the efficiency of both encodings, compared to each other, and to the Grew-match query system.

Our results show that existing off-the-shelf graph databases such as Neo4j can be very useful for performing large-scale complex syntactic searches in very large corpora.

1.1 Structure of the Paper

Section 2 gives an overview of UD treebanks, graph databases, and query languages. In section 3 we show two possible ways of encoding UD treebanks in a graph database, and in section 4 how to translate Grew-match queries into graph database queries. Section 5 consists of an evaluation of the two different treebank encodings, and in sections 6–7 we discuss the results and present some final conclusions.

[1]One such example is the research infrastructure Korp (Borin et al., 2012) from The Language Bank of Sweden, which contains more than 15 billion syntactically parsed and annotated tokens.

Proceedings of the 23rd International Workshop on Treebanks and Linguistic Theories (TLT, SyntaxFest 2025), pages 32–43

2 Background

2.1 UD Treebanks as Graphs

Conceptually, a UD treebank consists of sentences where every sentence is a graph. The nodes in a graph are the words in the sentence, and the dependency relations are labelled directed edges between the word nodes. Words are annotated with different attributes, such as *lemma*, *part-of-speech*, and *morphological features*. Sentence graphs are required to form a tree with one single word being the **`root`**, according to the annotation guidelines for universal dependencies (Zeman et al., 2023).

In addition to the strict tree structure, it is possible to add *enhanced* dependency relations to a sentence, which might turn the sentence into a general graph. In this paper we will not assume that a treebank only consists of trees, so the encoding that we introduce in section 3 will work on more generic "graph banks" as well as on UD treebanks.

The basic tokenisation level in UD is *syntactic* words, and not phonological or orthographic words. Some languages contract words, such as English "isn't" (*is not*) and German "im" (*in dem*), and this can be encoded in a UD treebank using *multi-word tokens*. Conceptually, we can think of this as special kind of node, spanning multiple words.

2.2 Searching in UD Treebanks

A common system for graph-based searches in treebanks is Grew-match. Searching is done by sending a request containing multiple items, describing constraints on graphs. The main item is specified using the keyword **`pattern`** and contains a list of clauses, describing the nodes returned by a search. The **`with`** keyword is used to introduce clauses without adding additional nodes to the result returned for a request. The **`without`** keyword describes negative constraints – only graphs that do *not* match these are returned for a request.

A *clause* in Grew-match is either a node or edge declaration, or an additional constraint.

- A node declaration `X [attr="val"]` describes a node named `X` having a property `attr` with the value `val`. All nodes represent words and properties represent the feature structures on these words.
- An edge declaration `X -[rel]-> Y` connects two nodes named `X` and `Y`. This declaration requires that there is a dependency of type `rel` from the word `X` to the word `Y`.
- Additional constraints can express a certain word order. Writing `X < Y` requires that the word `X` *immediately* precedes the word `Y`. Writing `X≪Y` requires that the word `X` occurs *somewhere* before the word `Y`.

Grew-match considers each sentence in a treebank as an individual graph and filters for those graphs matching the request.

2.3 Graph Databases

Neo4j is a general-purpose database system for graph-based data, similar to what a relational database is for tabular data. The system consists of a front-end in which you can formulate graph queries, a query-engine which plans and optimises the execution of queries, and a back-end storage system which handles persistence and data access. In contrast to Grew-match, Neo4j considers the whole database as a single graph.

The data model used by Neo4j is the *labelled property graph*, which represents data as a directed graph, where both nodes and edges may carry *labels* and attribute-value *properties*. This provides great flexibility and expressivity, as data can be represented in different nuanced ways.

In our encoding in section 3 we will use the labels mainly to specify the type of the node or label. Thus, when we write "a **`Word`** node", or "a **`SUCCESSOR`** edge", we actually mean "a node with the label **`Word`**", and "an edge with the label **`SUCCESSOR`**", respectively.

We adopt the convention that node labels are capitalised (e.g., **`Word`** and **`Sentence`**), but edge labels are uppercased (e.g., **`SUCCESSOR`** and **`DEPREL`**).

2.4 Cypher Query Language

The Cypher query language is used to query a Neo4j database (Francis et al., 2018). A query consists of multiple clauses, with the **`MATCH`**, **`WHERE`** and **`RETURN`** clauses being relevant for searching.

The **`MATCH`** clause introduces patterns to be matched in the graph. Writing `(n:Node {p:val})` in a **`MATCH`** clause describes a node named *n* labelled **`Node`** that has a property **`p`** with value `val`. An edge is written as `-[r:EDGE {p:val}]->` between two nodes. This example describes a directed edge labelled **`EDGE`** that has the property **`p`** with value `val` and is bound to the identifier *r*. In both cases, identifier, labels and properties are optional and may be omitted. The direction of an arrow describes the direction of a relationship.

```
ID  FORM    LEMMA   UPOS  XPOS  FEATS                                  HEAD  DEPREL
1   Surfen  Surfen  NOUN  NN    Gender=Neut|Number=Sing                   0   root
2-3 im      _       _     _     _                                         _   _
2   in      in      ADP   APPR  Case=Dat                                  4   case
3   dem     der     DET   ART   Case=Dat|Gender=Masc,Neut|Number=Sing     4   det
4   Garten  Garten  NOUN  NN    Gender=Masc|Number=Sing                   1   nmod
```

Figure 1: Example sentence (simplified) in CoNLL-U format from German-HDT (Borges Völker et al., 2019)

An edge suffixed with a plus character + indicates that two nodes are related via a sequence of edges matching the pattern (e.g. `-[:EDGE]->+` indicating a sequence of edges labelled **EDGE**).

The **WHERE** clause is used to additionally filter the matched subgraphs, using predicates that cannot be expressed by pattern matching. For example, properties of nodes and edges can be compared against each other or against regular expressions. In addition, there are keywords **EXISTS** and **NOT EXISTS** which have similar meaning as the **with** and **without** keywords in Grew-match.

Lastly, the **RETURN** clause is used to specify the result of a query. For every subgraph matched by a query, a record with all values specified in the **RETURN** clause will be returned.

While queries in Cypher are read from top to bottom with identifiers in later clauses being able to refer back to prior patterns, it is important to keep in mind that no order of execution is specified using queries. A database executing a query is free to reorder or simplify parts of the query in an attempt to optimize how it is executed, as long as the query result remains unchanged.

3 Encoding UD as Property Graphs

In order to store a UD-annotated treebank in a Neo4j database, it first has to be encoded as a labelled property graph. In this section we present an encoding scheme for dependencies, word annotations and structures, which we call the *property-based* encoding. Then we discuss an alternative encoding for annotations, which will be called the *node-based* encoding. Finally, we discuss how database constraints and indexes can be used to support the encoding.

3.1 Words and Dependencies

We encode each word as a node with label **Word**. A dependency between words is encoded as a **DEPREL** edge between the two corresponding **Word** nodes. The actual dependency relation is encoded as an edge property for the attribute **deprel**.

The root node in a sentence is encoded by labelling the corresponding **Word** node with the additional label **Root**.

3.2 Property-Based Encoding of Annotations

Figure 1 shows an example sentence in CoNLL-U format (Zeman et al., 2025).

The columns *ID*, *HEAD* and *DEPREL* (and *DEPS*, not shown in the example) are used to encode the (enhanced) dependency relation as **DEPREL** edges as discussed above, and therefore will not be encoded as node properties.

The columns *FORM*, *LEMMA*, *UPOS* and *XPOS* have a single value and will therefore each be encoded as single attributes to the **Word** node: **form**, **lemma**, **upos** and **xpos**, respectively. The column *FEATS* (and *MISC*, not shown in the example) contain attribute-value pairs. Each of these pairs will be encoded as an individual property.

As an example, wordline 3 in the example has a total of seven attributes to be encoded: *FORM=dem*, *LEMMA=der*, *UPOS=DET*, *XPOS=ART* are specified in separate columns, and we therefore straightforwardly encode them as properties on the corresponding **Word** node. The *FEATS* column specifies three morpohological features: *Case=Dat*, *Gender=Masc,Neut* and *Number=Sing*, which are encoded as additional properties on the **Word** node.

3.3 Sentences

Sentences are annotated with metadata and span multiple tokens. To make this metadata accessible, we need a way to encode sentences and associate them with spanned tokens. We do this by introducing a **Sentence** node for every sentence and encoding its metadata as properties on that node.

To associate words with their **Sentence** node, we create a **DEPREL** edge with **deprel=root** towards the root node. Then all words will be connected to their **Sentence** node by following the **DEPREL** edges.

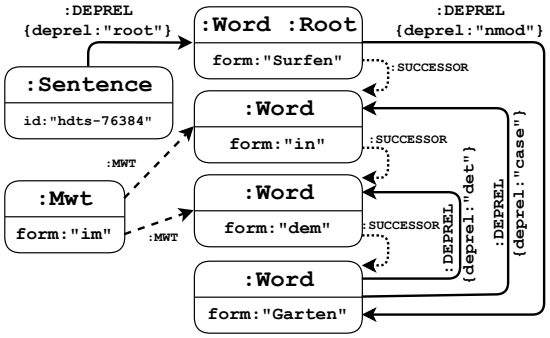

Figure 2: Example from Figure 1 encoded as a graph. Out of all properties, only **form** is shown.

To encode paragraphs or documents we follow the same strategy: create a **Paragraph** (or **Document**) node, and then create edges to its sentences (and edges from each document node to its paragraphs).

3.4 Linear Order

Words within a sentence are ordered, and a simple way to encode order in a graph database is via directed edges. Therefore, we introduce special edges to explicitly encode the word order within sentences.

For each word in a sentence we add a **SUCCESSOR** edge to its immediately succeeding word, except the final word which do not have a successor.

Figure 2 shows the example encoded as a labeled property graph, where **SUCCESSOR** edges are dotted. To reduce clutter only the **form** property is shown.

3.5 Multiword Tokens

The line "2–3" in Figure 1 is an example of a multiword token (MWT) – in German "im" is interpreted as a contraction of the syntactic words "in dem".

We encode multiword tokens in a similar way to sentences. For each multiword token, we add a new **Mwt** node, and **MWT** edges (dashed in Figure 2) from the node to each spanned word.

3.6 Alternative, Node-Based Encoding

In section 3.2 we showed how to encode attribute values as properties directly on the **Word** nodes.

An alternative strategy is to create a new node for each value of an attribute for a word, and add an edge from the **Word** node to the attribute node. For example, the attribute *Gender=Masc* would be represented as a **Gender** node, annotated with the property **value=Masc**. Note that we only create one such **Gender** node with the value **Masc**, and it will be shared by all **Word** nodes.

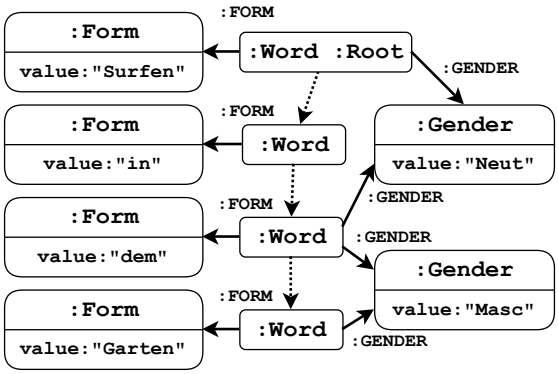

Figure 3: *Form* and *Gender* from Figure 1 encoded as a graph using the node-based encoding scheme.

There are multiple possible advantages of this encoding strategy. As following edges between nodes is a fast operation when querying a graph database, this encoding scheme could lead to better query performance. Further, by reusing nodes we aim to deduplicate data. Sections 5.1 and 5.4 show the impact on encoding size and query speed.

Further, wordline 3 in Figure 1 has a property *Gender=Masc,Neut*, meaning that the determiner can act as either masculine or neuter. This can be encoded as two **Gender** edges from the word, to the **Masc** and **Neut** nodes respectively. This can simplify some queries quite a lot, but we have not looked into this because it is not needed for our translation from the Grew-match query language.

Figure 3 shows the node-based encoding of the example in Figure 1. To reduce clutter only the **Form** and **Gender** nodes are shown.

The node- and the property-based encodings are freely interchangeable. The encoding of dependency relations, sentences and MWTs remains unchanged when choosing between the node-based and the property-based encoding strategy. It is even possible to mix both strategies, encoding only some attributes as nodes and others as properties.

3.7 Constraints and Indexes

Adding constraints and indexes helps the query planner to improve query performance. Uniqueness constraints in Neo4j ensure that a combination of label and property value appears only once in the database. Similarly, indexes can be used to quickly find nodes or edges with a combination of label and property value.

We add a uniqueness constraint and index for every attribute encoded using the node-based strategy. This should improve performance for queries

on the node-based encoding. Additionally, we add another index on **DEPREL** edges and their **deprel** property, which enables fast lookup of nodes connected by an edge in the index.

4 Querying in Encoded Corpora

To demonstrate the capabilities of the Cypher query language, we will now describe a straightforward algorithmic approach to translate Grew-match queries into Cypher queries.

For every word matched in a Grew-match **pattern**, add a **MATCH** clause for a **Word** node with the same identifier. Further, add a single **RETURN** clause at the end of the query and add all introduced identifiers to that clause. While it is not necessary to match all words first, doing so simplifies the translation, as words can be referred to by their identifier afterwards. Then, translate each of the clauses in a Grew-match request as follows:

- An edge clause specifying a dependency relation `X -[aux]-> Y`, is translated into a **MATCH** clause specifying the same edge `(X)-[:DEPREL {deprel:"aux"}]->(Y)`.
- Clauses specifying immediate precedence (written `X < Y`), are translated into a **MATCH** clause with an edge `(X)-[:SUCCESSOR]->(Y)`. General precedence between nodes (written `X≪Y`) is translated similarly, allowing the two nodes to be related via a sequence of edges `(X)-[:SUCCESSOR]->+(Y)`.
- **with** clauses are translated as if they were part of a **pattern**, adding an initial **MATCH** clause for all occurring words and translating each clause. Identifiers for these words must not be included in the **RETURN** clause and possibly require renaming if they occur in multiple **with** patterns.
- **without** clauses are translated into a **NOT EXISTS** expression as part of the **WHERE** clause containing a translation of the individual Grew-match clauses.

Clauses in Grew-match accessing annotations have to be translated differently depending on the encoding scheme. We will consider how this is done on the example `X [upos="NOUN"]`.

- If encoded as properties, a **MATCH** clause is added specifying identifier and the requested properties: `(X {upos:"NOUN"})`.
- If encoded as nodes, a **MATCH** clause is added for an edge between the word node and the node representing the requested value: `(X)-[:UPOS]->(:Upos {value:"NOUN"})`.

  If multiple attribute-value pairs are specified, a **MATCH** clause is added for each of them.

Values of properties in Grew-match can also be specified in terms of a regular expression or a disjunction of values. In these cases a direct translation into a **MATCH** clause is not possible and we use the **WHERE** clause to represent these constraints. For the node-based encoding of annotations, the corresponding nodes have to be fetched via a **MATCH** clause without specifying their value. A Grew-match clause like `X [lemma="der"|"die"]` therefore turns into `WHERE X.lemma IN ["der","die"]`, under the property-based annotation scheme. For the node-based annotation scheme it is instead translated into the following two clauses: `MATCH (X)-[:LEMMA]->(xlemma:Lemma)`, and `WHERE xlemma.value IN ["der","die"]`.

There are some special cases to consider when translating queries. A query for the root node of a sentence can be translated into a **MATCH** clause with the **Root** label. Further, queries in Grew-match consider each sentence as an individual graph. Consequently, all words in a Grew-match request are implicitly constrained to the same sentence. This restriction has to be translated as well, by adding a **MATCH** clause relating otherwise unrelated words to the same dependency tree: `(X)-[:DEPREL]-+(Y)`.

5 Evaluation

This section presents the performance of Neo4j as a corpus system for UD-annoated treebanks in different scenarios to evaluate, (a) whether Neo4j is a viable system for treebanks and (b) how the presented encoding schemes perform. We mainly consider two perspectives here: The perspective of an administrator encoding treebanks and provisioning the storage space for the database. And the perspective of users wanting to search treebanks with quick query response times. Our evaluations cover encoding time, required disk space for encoded treebanks and query execution time.

We developed a tool to automatically encode and import UD-annotated treebanks into a Neo4j database using the presented encoding schemes. The tool and its source code is freely accessible online.[2] Our tool accepts CoNLL-U files as its

[2]Source code, executables and experiment data is available at https://github.com/Niklas-Deworetzki/neo4j-ud-importer

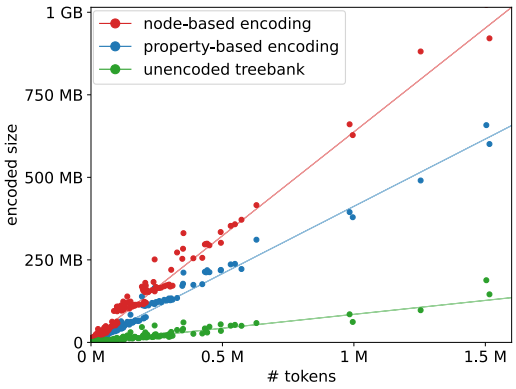

Figure 4: Disk size of treebanks.

input, encodes the treebank described by these files as a graph and stores this graph in a Neo4j database.

Our measurements were obtained using Neo4j version 5.26.0 and Grew-match version 1.16.1, both running in Docker on the same hardware.

5.1 Encoding Corpora

We automatically encoded all treebanks in UD release 2.15 (Zeman et al, 2024) to measure time and disk space requirements. Figure 4 shows the required disk size of treebanks in bytes in relation to treebank size in tokens.[3] The disk size of an encoded treebank is calculated as the sum of the size of all files in the `database` directory in Neo4j. A clear linear relationship between disk size and treebank size can be seen. The encoding time also has a strong linear relationship with treebank size – it takes around 1 minute for the property-based encoding to encode a 1-million token treebank, and 2–3 minutes for the node-based encoding. The property-based encoding requires approximately 6 times as much disk space as the CoNLL-U files, while the node-based encoding requires approximately 10 times as much disk space.

5.2 Benchmarking Setup

To measure the performance of Neo4j as a query system for UD-annoated treebanks, we translated and ran a set of queries from Weissweiler et al. (2024). They present rules for the Grew-match graph rewriting framework to automatically annotate constructions in UD treebanks. We selected queries from these rules for four different constructions present in ten different languages (namely interrogatives, existentials, conditionals and NPN – a repeated noun with an adposition in between).

Important for our selection is that the chosen requests cover a variety of languages, cover many aspects of the Grew-match query language, and are relevant for linguistic research. Details of these queries are not important to our evaluation, but are explained further in Appendix C. We used the procedure from section 4 to translate the four chosen patterns for each of the different languages into equivalent Cypher queries for both encoding variants, resulting in a total of 80 translated queries.

To execute queries and measure their execution time, we grouped queries for the same language and encoding scheme together. The four queries were executed in sequence, and the sequence was repeated multiple times. The goal of this is to increase cache pressure, so that it is not possible for a query system to simply "remember" the results for one particular query. We then started up a server with one encoded corpus, executed the sequence of queries for that corpus 10 times to warm up caches, and then took measurements by repeating queries in sequence 100 times. For each query, we recorded the median of all 100 collected execution times. Queries for Neo4j were sent to the database server for execution, while queries for Grew-match were executed by accessing its command line interface.

We selected the biggest available corpus for each of the ten languages to run our measurements on. The complete list of languages and used corpora can be found in Appendix A.

5.3 Comparing Neo4j and Grew-match

Neo4j using the property-based encoding requires on average 1% of query execution time compared to Grew-match. More precisely, Grew-match requires between 30 (for the NPN query on the Hindi treebank) and 600 (for conditionals in Portugese and existentials in Spanish) times as much execution time. There is, however, one exception: for the interrogatives in Hindi, Neo4j was actually slower, requiring 10% more execution time.

On all measured systems, the execution time is roughly proportional to the size of the queried treebank. Per million tokens of treebank size, Grew-match requires on average 28 seconds of query time, while Neo4j requires 0.28 seconds (for the property-based encoding) and 0.31 seconds (for the node-based encoding). A table listing all execution times is shown in Appendix B.

[3] The figure omits the Hamburg Dependency Treebank, which with 3.4 million tokens lies far outside the shown range and approximately 5% below the trend line.

Language	Cond.	Exist.	Interrog.	NPN
Chinese	0.24	0.72	0.25	1.60
Coptic	0.67	0.85	0.66	1.94
English	0.55	0.52	0.70	2.64
French	0.59	1.05	**5.45**	1.24
German	0.64	0.58	0.08	1.33
Hebrew	0.62	0.78	**5.99**	0.98
Hindi	0.27	0.39	1.07	**6.76**
Portuguese	0.88	0.57	0.21	1.57
Spanish	0.51	0.74	0.16	1.47
Swedish	0.67	0.50	0.74	1.52
Average	0.53	0.64	0.35	1.53

Table 1: Execution time of the node-based encoding, relative to the property-based encoding, per query type. A value of 0.25 means that the node-based encoding is 4 times faster. Outliers are **bold-faced**, and they are *not* included in the calculation of the average speed-up. Because the values are factors, the average is calculated as the geometric mean.

5.4 Comparing Encoding Strategies

A comparison of the query execution times for the property-based and node-based encoding is shown in Table 1. The results in that table show groups of similar relative execution times: In general, the node-based encoding is faster, requiring 40% to 80% of execution time for most queries. For interrogatives in Hindi and NPN's in Hebrew, there is no difference between both encoding schemes. For all of the NPN queries, the node-based encoding is actually slower, requiring 1.5 times longer execution time on average. For interrogatives in French and Hebrew, as well as NPN's in Hindi, the node-based encoding is 5–7 times slower. On the other hand, it is 4–10 times faster for conditionals in Hindi and Chinese, and for interrogatives in Chinese, German, Portuguese and Spanish.

5.5 Execution Time and Corpus Size

To better understand how execution times scale with respect to corpus size, we ran the same set of queries on differently-sized subsets of the Hamburg Dependency Treebank. These sub-corpora were obtained by randomly sampling $10\%, 20\%, \ldots, 90\%$ of sentences from the original corpus.

We used the same setup as presented in Section 5.2 to execute all four queries for the German language on these corpora. The measured execution times are shown in Figure 5 and show a clear linear relationship between execution time and corpus size for each of the executed queries.

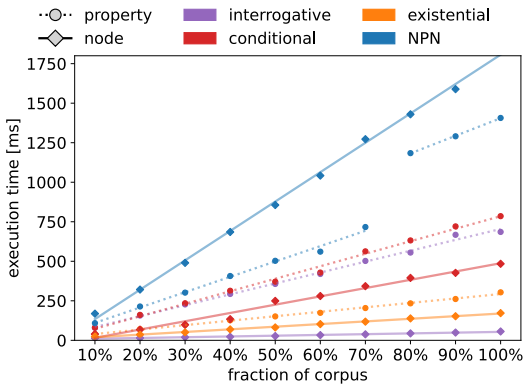

Figure 5: Execution time for differently sized subcorpora of HDT. Note the discontinuity for NPN.

There is one exception to this linear dependency, which is seen in the diagram: there is an unexpected jump for NPN queries in the property-based encoding after 70% of the corpus size.

6 Discussion

Our experiments clearly show that Neo4j is a viable corpus search system for UD treebanks.

6.1 Comparing Neo4j and Grew-match

On average, Neo4j outperforms Grew-match in almost all cases by orders of magnitude. Most queries run in fractions of a second where Grew-match needs several seconds for the same query.

We believe that one reason for this improvement is that Neo4j considers the whole treebank as one single graph, and therefore it can make use of search indexes (as discussed in section 3.7). Grew-match on the other hand considers every sentence to be a separate graph, which makes it much harder to do global optimisations, and therefore it has to test each sentence against the query iteratively.

The main trade-off with using Neo4j instead of Grew-match is disk usage. Figure 4 shows that encoding the treebank in a graph database uses 6 – 10 times more space than the original CoNLL-U text files, depending on the encoding.

For example, all UD treebanks combined consist of roughly 32M tokens, or 2.9 GB of CoNLL-U text files. In comparison, the Neo4j database files require 14 or 22 GB (depending on the encoding), which is a considerable overhead, but still manageable.

6.2 Hindi Interrogatives

There is one notable exception, the interrogative query in Hindi shows no improvement at all com-

pared to Grew-match. This is the case both for the node-based and the property-based encoding. The query itself consists of a disjunction of several lemmas, followed by a filter that rules out sentences containing some subtrees that are unrelated to the actual lemma. Because of this, Neo4j has to add additional constraints to make sure that the subtrees are in the same sentence as the lemma it is searching for, so it has to do extra work and cannot make use of the global indexes. See Appendix C for the query and its translation.

We did try a simple optimisation in our encodings, where we created direct edges from word nodes to their sentence nodes. This improved the execution time for Hindi interrogatives (and some other queries) by up to 100 times. We did not perform any in-depth evaluation of this and other possible optimisations, but it suggests that it is possible to improve the corpus encoding substantially if we know what kind of queries we will perform.

6.3 Comparing Encodings

The size of the encoded corpora grow linearly in the size of the treebanks, and the property-based encoding requires only around 60% as much storage as the node-base encoding. Extracting different values into separate, shared nodes provides no benefit in terms of storage size. The reason for this is that Neo4j stores string values not as part of nodes, but in a separate unit (Rocha, 2020). Therefore, strings occurring multiple times in the dataset will result in only one copy stored in the database with multiple references to that one copy.

In terms of execution time, the node-based encoding is usually faster than the property-based, by a factor of 1.5–3. But this is not always the case: for the NPN queries it is the property-based encoding that is faster by a factor of 1.5–2.5. And there are some few extreme outliers, where the property-based encoding is actually 6–7 times faster.

Looking into the execution plans and profiling information for these queries suggests that having each attribute as a separate node in the graph is the reason for both of these behaviors. In situations where the node-based encoding outperforms the property-based one, it does so by making use of uniqueness constraints and indexes. For example, one lookup in the **POS**-index will yield the node for a certain part-of-speech, which has a reference to all matching words. The property-based representation on the other hand, linearly scans through words to find nodes for which relations can be resolved and further constraints checked. When it comes to the NPN construction queries, this linear scan is advantageous. The query asks for three subsequent tokens, and the database has the successors readily available when we use the property-based encoding. For the node-based encoding, the database opts to find all words related to the single **Noun** node, followed by subsequently finding their successors and their part-of-speech nodes. This results in many accesses to the underlying storage at many different positions, resulting in slow execution times.

The conclusion from this is, that the node-based encoding can make use of available indices and uniqueness constraints efficiently, outperforming the property-based encoding for most queries. However, this is not true in all cases, and the relative simplicity of the property-based encoding sometimes results in lower execution times, as shown by the NPN queries.

6.4 Scalability

Comparing the same query on differently sized sub-corpora we see that the execution time grows linearly in the size of the corpus size for all queries and encoding strategies. We do not know why the property-based encoding experiences a bigger-than-expected jump in execution time for NPN queries when going from 70% to 80% of the original corpus size. Maybe it has to do with caching of intermediate results and that the system runs out of internal memory, but that is just a guess.

Encoding the corpus in Neo4j seems to improve the search speed by around 100 times on average, compared to Grew-match. Therefore we draw the conclusion that it should be feasible to use any of our encodings on treebanks with 100 million tokens or more. Such a treebank would require about 100 GB of storage space, which is feasible on modern computers.

Note that we got these improvements despite using a very simplistic encoding of the treebanks into a graph database. As suggested by our optimisation in section 6.2 there is probably a lot of opportunities for further improvement.

6.5 Use as a Corpus System

One thing we have not looked into in this study is how to incorporate Cypher and Neo4j in a full-fledged corpus system, such as Grew. Grew-match is just one part of the Grew system, which is a general graph-rewriting framework with which one can create, annotate, and update treebanks. In addition

to searching within a treebank, nodes and edges can be created or deleted, nodes can be re-ordered and annotations can be changed. Cypher supports similar functionality via commands such as **CREATE**, **DELETE**, and **SET**, for modifying the database in different ways. More work would be required to map different Grew commands to these Cypher clauses.

7 Conclusion

Our main conclusion from this evaluation is that graph databases are viable as backend storage for treebanks. The study is only done on UD treebanks, but there is nothing very UD-specific in our encodings or the graph databases, so we believe that this would be useful for all kinds of treebanks.

Using graph databases it will be possible to search for complex syntactic phenomena in very large treebanks with 100 million tokens and more.

Since we translate the treebanks to a general graph, it should definitely be possible to include more kinds of relations, such as anaphoric references, semantic databases, and morphological segmentation. Including all kinds of relations in one single graph database opens up for doing large-scale searching for complex queries on several linguistic levels at once.

Acknowledgments

We thank Nicholas Smallbone and three anonymous reviewers for valuable discussions, comments and insights.

References

Riyaz Ahmad Bhat, Rajesh Bhatt, Annahita Farudi, Prescott Klassen, Bhuvana Narasimhan, Martha Palmer, Owen Rambow, Dipti Misra Sharma, Ashwini Vaidya, Sri Ramagurumurthy Vishnu, et al. 2017. The Hindi/Urdu treebank project. In *Handbook of Linguistic Annotation*. Springer Press.

Guillaume Bonfante, Bruno Guillaume, and Guy Perrier. 2018. *Application of Graph Rewriting to Natural Language Processing*. Wiley Online Library.

Emanuel Borges Völker, Maximilian Wendt, Felix Hennig, and Arne Köhn. 2019. HDT-UD: A very large Universal Dependencies treebank for German. In *Proceedings of the Third Workshop on Universal Dependencies (UDW, SyntaxFest 2019)*, pages 46–57, Paris, France. Association for Computational Linguistics.

Lars Borin, Markus Forsberg, and Johan Roxendal. 2012. Korp – the corpus infrastructure of Språkbanken. In *Proceedings of the 8th International Conference on Language Resources and Evaluation (LREC'12)*, pages 474–478, Istanbul, Turkey. European Language Resources Association (ELRA).

António Branco, João Ricardo Silva, Luís Gomes, and João António Rodrigues. 2022. Universal grammatical dependencies for Portuguese with CINTIL data, LX processing and CLARIN support. In *Proceedings of the 13th International Conference on Language Resources and Evaluation (LREC'22)*, pages 5617–5626, Marseille, France. European Language Resources Association (ELRA).

Marie-Catherine de Marneffe, Christopher D. Manning, Joakim Nivre, and Daniel Zeman. 2021. Universal Dependencies. *Computational Linguistics*, 47(2):255–308.

Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 44th International Conference on Management of Data (SIGMOD'18)*, page 1433–1445, Houston, USA. Association for Computing Machinery (ACM).

GSDSimp. 2023. Simplified chinese universal dependencies. Accessed 2025-04-14, https://github.com/UniversalDependencies/UD_Chinese-GSDSimp.

Bruno Guillaume. 2021. Graph matching and graph rewriting: GREW tools for corpus exploration, maintenance and conversion. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics (EACL): System Demonstrations*, pages 168–175, Online. Association for Computational Linguistics (ACL).

Bruno Guillaume, Marie-Catherine de Marneffe, and Guy Perrier. 2019. Conversion et améliorations de corpus du français annotés en Universal Dependencies [conversion and improvement of Universal Dependencies French corpora]. *Traitement Automatique des Langues*, 60(2):71–95.

Peter Kleiweg and Gertjan van Noord. 2020. AlpinoGraph: A graph-based search engine for flexible and efficient treebank search. In *Proceedings of the 19th International Workshop on Treebanks and Linguistic Theories*, pages 151–161, Düsseldorf, Germany. Association for Computational Linguistics.

Thomas Krause and Amir Zeldes. 2014. ANNIS3: A new architecture for generic corpus query and visualization. *Digital Scholarship in the Humanities*, 31(1):118–139.

Taulé Mariona, M. Antònia Martí, and Marta Recasens. 2008. AnCora: Multilevel Annotated Corpora for Catalan and Spanish. In *Proceedings of the 6th International Conference on Language Resources and Evaluation (LREC'08)*, pages 96–101, Marrakech, Morocco. European Language Resources Association (ELRA).

Joakim Nivre, Jens Nilsson, and Johan Hall. 2006. Talbanken05: A Swedish treebank with phrase structure and dependency annotation. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC'06)*, Genoa, Italy. European Language Resources Association (ELRA).

Martha Palmer, Rajesh Bhatt, Bhuvana Narasimhan, Owen Rambow, Dipti Misra Sharma, and Fei Xia. 2009. Hindi syntax: Annotating dependency, lexical predicate-argument structure, and phrase structure. In *The 7th International Conference on Natural Language Processing*, pages 14–17.

José Rocha. 2020. Understanding Neo4j's data on disk. Accessed 2025-04-14, https://neo4j.com/developer/kb/understanding-data-on-disk/.

Shoval Sade, Amit Seker, and Reut Tsarfaty. 2018. The Hebrew Universal Dependency treebank: Past, present and future. In *Proceedings of the Second Workshop on Universal Dependencies (UDW 2018)*, pages 133–143, Brussels, Belgium. Association for Computational Linguistics.

Natalia Silveira, Timothy Dozat, Marie-Catherine de Marneffe, Samuel Bowman, Miriam Connor, John Bauer, and Chris Manning. 2014. A gold standard dependency corpus for English. In *Proceedings of the 9th International Conference on Language Resources and Evaluation (LREC'14)*, pages 2897–2904, Reykjavik, Iceland. European Language Resources Association (ELRA).

Jan Štěpánek and Petr Pajas. 2010. Querying diverse treebanks in a uniform way. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, Valletta, Malta. European Language Resources Association (ELRA).

Leonie Weissweiler, Nina Böbel, Kirian Guiller, Santiago Herrera, Wesley Scivetti, Arthur Lorenzi, Nurit Melnik, Archna Bhatia, Hinrich Schütze, Lori Levin, Amir Zeldes, Joakim Nivre, William Croft, and Nathan Schneider. 2024. UCxn: Typologically informed annotation of constructions atop Universal Dependencies. In *Proceedings of the Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING'24)*, pages 16919–16932, Torino, Italia. European Language Resources Association (ELRA).

Amir Zeldes and Mitchell Abrams. 2018. The Coptic Universal Dependency treebank. In *Proceedings of the Second Workshop on Universal Dependencies (UDW 2018)*, pages 192–201, Brussels, Belgium. Association for Computational Linguistics.

Daniel Zeman, Joakim Nivre, Nathan Schneider, Filip Ginter, and Christopher Manning. 2023. Universal dependencies guidelines. Accessed 2025-04-14, https://universaldependencies.org/guidelines.html.

Daniel Zeman, Joakim Nivre, Nathan Schneider, Filip Ginter, and Sampo Pyysalo. 2025. CoNLL-U format. Accessed 2025-04-14, https://universaldependencies.org/format.html.

Daniel Zeman et al. 2024. Universal Dependencies 2.15. LINDAT/CLARIAH-CZ digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

A List of Treebanks used for Execution Time Measurements

The complete list of all treebanks used for our execution time measurements is shown in Table 2. The collection of languages is determined by the languages for which Weissweiler et al. (2024) provide automated annotation rules. We chose the largest available treebank for each of these languages.

B Table of Execution Time Measurements

Table 3 contains the results of our benchmark, showing all 120 data points obtained from running queries for 4 different constructions in 10 different languages on 3 query systems. The order of languages presented in this table follows Table 2.

C List of Grew-match and Neo4j Queries

A selection of queries used for our execution time measurements is shown in Table 4. The complete list of queries is available online: https://github.com/Niklas-Deworetzki/neo4j-ud-importer/tree/main/experiments/queries

Lang.	Treebank	# Tokens
German	HDT (Borges Völker et al., 2019)	3,399,390
Spanish	AnCora (Mariona et al., 2008)	547,558
Portuguese	CINTIL (Branco et al., 2022)	441,991
French	GSD (Guillaume et al., 2019)	389,367
Hindi	HDTB (Bhat et al., 2017; Palmer et al., 2009)	351,704
English	EWT (Silveira et al., 2014)	251,493
Chinese	GSDSimp (GSDSimp, 2023)	123,291
Hebrew	HTB (Sade et al., 2018)	114,648
Swedish	Talbanken (Nivre et al., 2006)	96,820
Coptic	Scriptorium (Zeldes and Abrams, 2018)	26,837

Table 2: UD treebanks used for our benchmark ordered by size.

Lang.	Query	Grew	Prop.	Node	Lang.	Query	Grew	Prop.	Node
German	cond.	70.84	0.714	0.458	**Spanish**	cond.	15.05	0.059	0.030
	exist.	67.49	0.283	0.165		exist.	14.95	0.025	0.019
	interrog.	67.43	0.643	0.050		interrog.	15.33	0.126	0.020
	NPN	68.02	1.282	1.699		NPN	15.05	0.179	0.262
Portugese	cond.	8.52	0.014	0.012	**French**	cond.	7.18	0.035	0.021
	exist.	8.50	0.103	0.059		exist.	7.16	0.019	0.020
	interrog.	8.48	0.089	0.019		interrog.	7.26	0.081	0.441
	NPN	8.57	0.135	0.212		NPN	7.21	0.151	0.188
Hindi	cond.	16.16	0.109	0.029	**English**	cond.	5.79	0.049	0.027
	exist.	16.24	0.224	0.086		exist.	5.77	0.064	0.033
	interrog.	16.22	17.633	18.906		interrog.	5.84	0.028	0.020
	NPN	16.19	0.028	0.192		NPN	6.00	0.043	0.112
Chinese	cond.	3.54	0.033	0.008	**Hebrew**	cond.	3.11	0.027	0.017
	exist.	3.53	0.020	0.014		exist.	3.09	0.017	0.013
	interrog.	3.52	0.029	0.007		interrog.	3.11	0.020	0.119
	NPN	3.54	0.041	0.065		NPN	3.12	0.094	0.092
Swedish	cond.	2.43	0.031	0.021	**Coptic**	cond.	1.28	0.021	0.014
	exist.	2.42	0.031	0.015		exist.	1.27	0.011	0.009
	interrog.	2.45	0.019	0.014		interrog.	1.29	0.021	0.014
	NPN	2.43	0.042	0.063		NPN	1.32	0.017	0.033

Table 3: Execution times measured in seconds for the three query systems (**Grew**-match, Neo4j with **prop**erty-based encoding and Neo4j with **node**-based encoding) on equivalent queries ordered by language and construction.

Query	Grew-match	Property-based	Node-based

German exist.	pattern {
	E[lemma="es"];
	G[lemma="geben"];
	G-[nsubj]->E;
	}
		MATCH (E:Word)
		MATCH (G:Word)
		MATCH (E {LEMMA:'es'})
		MATCH (G {LEMMA:'geben'})
		MATCH (G)-[:DEPREL
		{deprel:'nsubj'}]->(E)
		RETURN E, G
			MATCH (E:Word)
			MATCH (G:Word)
			MATCH (E)-[:LEMMA]->
			(:Lemma {value:'es'})
			MATCH (G)-[:LEMMA]->
			(:Lemma {value:'geben'})
			MATCH (G)-[:DEPREL
			{deprel:'nsubj'}]->(E)
			RETURN E, G

Hindi interrog.	pattern {
	W [lemma="क्या"
	|"कौन"
	|"कहां"
	|"कब"
	|"कैसे"
	|"कितना"
	|"किस"];
	} without {
	SC [form="कि"];
	V -[mark]-> SC
	} without {
	V1 [upos=VERB];
	V1 -[advcl]-> V;
	}
		MATCH (W:Word)
		WHERE W.LEMMA in [...]
		AND NOT EXISTS {
		MATCH (SC:Word)
		MATCH (V:Word)
		MATCH (V)-[:DEPREL]-+(W)
		MATCH (SC FORM:'कि')
		MATCH (V)-[:DEPREL
		{deprel:'mark'}]->(SC)
		} AND NOT EXISTS {
		MATCH (V1:Word)
		MATCH (V:Word)
		MATCH (V1)-[:DEPREL]-+(W)
		MATCH (V1 UPOS:'VERB')
		MATCH (V1)-[:DEPREL
		{deprel:'advcl'}]->(V)
		} RETURN W
			MATCH (W:Word)
			MATCH (W)-[:LEMMA]->
			(wlemma:Lemma)
			WHERE wlemma.value in [...]
			AND NOT EXISTS {
			MATCH (SC:Word)
			MATCH (V:Word)
			MATCH (V)-[:DEPREL]-+(W)
			MATCH (SC)-[:FORM]->
			(:Form {value:'कि'})
			MATCH (V)-[:DEPREL
			{deprel:'mark'}]->(SC)
			} AND NOT EXISTS {
			MATCH (V1:Word)
			MATCH (V:Word)
			MATCH (V1)-[:DEPREL]-+(W)
			MATCH (V1)-[:UPOS]->
			(:Upos {value:'VERB'})
			MATCH (V1)-[:DEPREL
			{deprel:'advcl'}]->(V)
			} RETURN W

Chinese NPN	pattern {
	N1 [upos=NOUN];
	P [upos=ADP];
	N2 [upos=NOUN];
	N1 < P;
	P < N2;
	N1.form=N2.form;
	}
		MATCH (N1:Word)
		MATCH (P:Word)
		MATCH (N2:Word)
		MATCH (N1 {UPOS:'NOUN'})
		MATCH (N2 {UPOS:'NOUN'})
		MATCH (P {UPOS:'ADP'})
		MATCH (N1)-[:SUCCESSOR]->(P)
		MATCH (N2)<-[:SUCCESSOR]-(P)
		WHERE N1.FORM = N2.FORM
		RETURN N1, N2, P
			MATCH (N1:Word)
			MATCH (P:Word)
			MATCH (N2:Word)
			MATCH (N1)-[:UPOS]->
			(:Upos {value: 'NOUN'})
			MATCH (N2)-[:UPOS]->
			(:Upos {value: 'NOUN'})
			MATCH (P)-[:UPOS]->
			(:Upos {value: 'ADP'})
			MATCH (N1)-[:SUCCESSOR]->(P)
			MATCH (N2)<-[:SUCCESSOR]-(P)
			WHERE (N1)-[:FORM]->(:Form)
			<-[:FORM]-(N2)
			RETURN N1, N2, P

Table 4: A sample of queries used for execution time measurements. The table shows a Grew-match pattern and the translated Cypher queries, following the translation scheme provided in Section 4. Note that the full list of lemmas for the Hindi interrogative query is only shown for Grew-match and is abbreviated for the other two columns.