# Neo4j

Nico-Alexei Hein
it12125@lehre.dhbw-stuttgart.de

March 14, 2015

**Abstract**

This is a rough explanation of Neo4j internals with a focus on disk storage and file layout.

## 1 Introduction

Neo4j is a graph based database.

### 1.1 Cypher

Neo4j has its own query language which is called Cypher. Cypher concentrates on pattern search.

### 1.2 Example

As an example we modelled the relation between books, authors and publishers. Authors write books and publisher publish books. To start a database, Neo4j has to be run from the command line or terminal. After the server did start, a graphical user interface can be accessed through a browser. This graphical user interface has its own command line in which Cypher statements are executed.

### 1.3 Syntax

The syntax of Cypher statements are similar to SQL statements. Figure 1 Cypher Syntax shows the basic structure of a statement.

```
1  MATCH <pattern> WHERE <conditions> RETURN <expressions>
```

Listing 1: Cypher Syntax

**CREATE** The CREATE statement can be used to create new nodes as well as create relations between notes. Figure 2 Create Node shows how to create the author Fitzgerald. The CREATE statement is started with parenthesis to indicate a node. The first information given to the CREATE statement is the ID of the node, followed by a Doppelpunkt and a lable given to the node. In this case the ID is fscottfitzgerald and the label for the node is Author. All properties follow in brakets. The node here has three attributes: name, first-name and stage-name.

```
1  CREATE (fScottFitzgerald:Author { name : 'Fitzgerald', firstname : 'Francis Scott Key',
       stagename : 'F. Scott Fitzgerald'})
```

Listing 2: Create Node

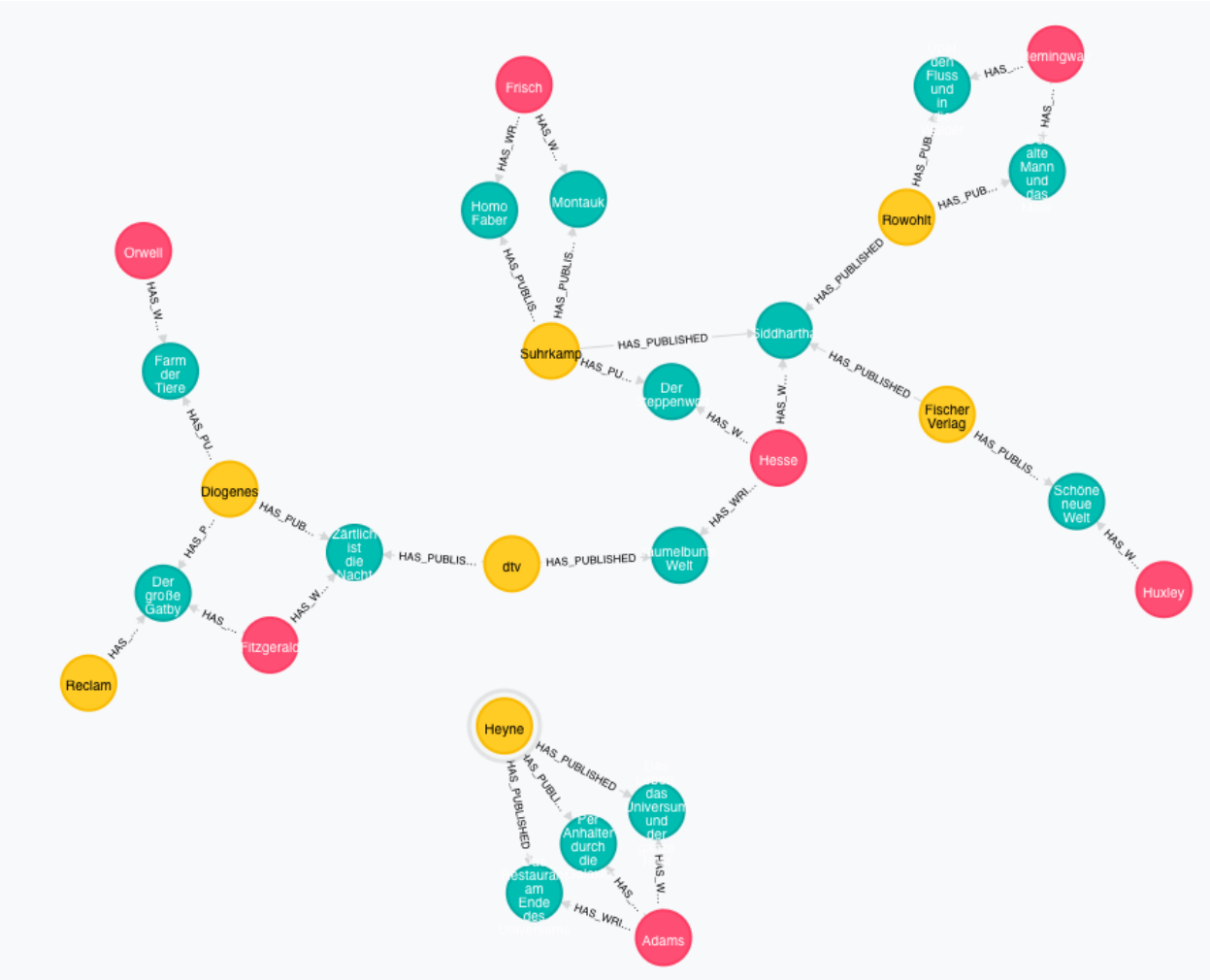



Figure 1: Example Graph

The statement for creation of a relation looks a bit different. Figure 3 Create Relation shows an example where the relation between Fitzgerald and his book is made. The node ID from which the relations starts is put in parenthesis at the beginning. Then the kind of relation is defined with. Beginning with a Doppelpunkt the relationship type is defined, followed by all properties in brakets. After the relationship information are finished an arrow points to the end node, again defined by the node ID in parenthesis.

```
1  CREATE (fScottFitzgerald)-[:HAS_WRITTEN { year : '1925'}]->(derGrosseGatsby)
```

Listing 3: Create Relation

**MATCH**

**DELETE**

### 1.4 Solving

## 2 Storage Structure

### 2.1 Graph representation of the Data

The simple sample graph displayed in figure 2 shows a subgraph of the Neo4j example. This human readable representation is now used to explain how Neo4j stores this graph.

### 2.2 Property Records

As a first step we take a look at the data stored in this graph. In Neo4j it is possible to store properties for every node and relationship (labels are ignored ignored at the moment). To store the nodes and relationships itself in fixed size records (for faster access etc.) linked lists are used. In figure 3 the property records are displayed.

Every node and every relationship references its first property record. The property records themselves can be seen as a double linked list with a key - value store. In figure 9 the representation on byte-level is shown. Node, relationship and property records are storing pointer to the next property in the last 4 bytes (one integer in Java). Bytes 18 to 21 of property record are containing a pointer to the previous record.

### 2.3 Node Records

The nodes itself as well as relationships are stored in fixed size records. However a node needs to know all relationships it is involved in. Therefor again a linked list is used. Every node references its first relationship shown in figure 4 and in figure 9 the deep blue marked bytes. Now the complete node recored was covered and before we are focusing on the double linked relationship lists the relationship record is explained.

### 2.4 Relationship Records

As could have been observed above the node record only stores a pointer to a property and a relationship but not the first adjacent. Therefore we change our perspective to the graph and are focusing on relationships. Instead of nodes referencing adjacent nodes we have nodes referencing the first relationship which references the start-node and the end-node of the relationship (Grey filled bytes in figure 9). This results in the abstract graph shown in figure 5.

The relationship type (Blue in figure 5) references a relationship type which again references a string (with the name) in the dynamic store. (This avoids redundancy in storage - the cache works in a different way.)

The four green integers of the relationship record (figure 9) are used to realize the mentioned double linked relationship lists.

Since every relationship is involved into two double linked lists (the one of the start-node and the one of the end-node) four pointers need to be stored. In figure 5. They are shown as:

- **SP** Start-Node Previous Relationship
- **EP** End-Node Previous Relationship
- **SN** Start-Node Next Relationship
- **EN** End-Node Next Relationship

In figure 6 only the SP is set. In the following list some pointers are explained:

- 'Frisch - has written - HomoFaber' has no pevious relationship since Frisch references it as its first relationship.
- 'Frisch - has written - Montauk' has 'Frisch - has written - HomoFaber' as previous relationship.
- 'Suhrkamp - has published - Steppenwolf' has 'Suhrkamp - has published - Montauk' which has 'Suhrkamp - has published - HomoFaver' which has no previous relationship.
- ...

In figure 7 the links for the end-node previous relationship are added analogue to start-node previous relationship.

Now in figure 8 all double linked lists are complete. This is how the graph displayed in figure 2 would be stored.

## 2.5 Sources

- Slides from http://slideshare.net/thobe/an-overview-of-neo4j-internals by Tobias Lindaaker (assumed to be reliable since http://www.neo4j.org/develop/internals revers to these slides)
- http://digitalstain.blogspot.de/2010/10/neo4j-internals-file-storage.html by Chris Gioran

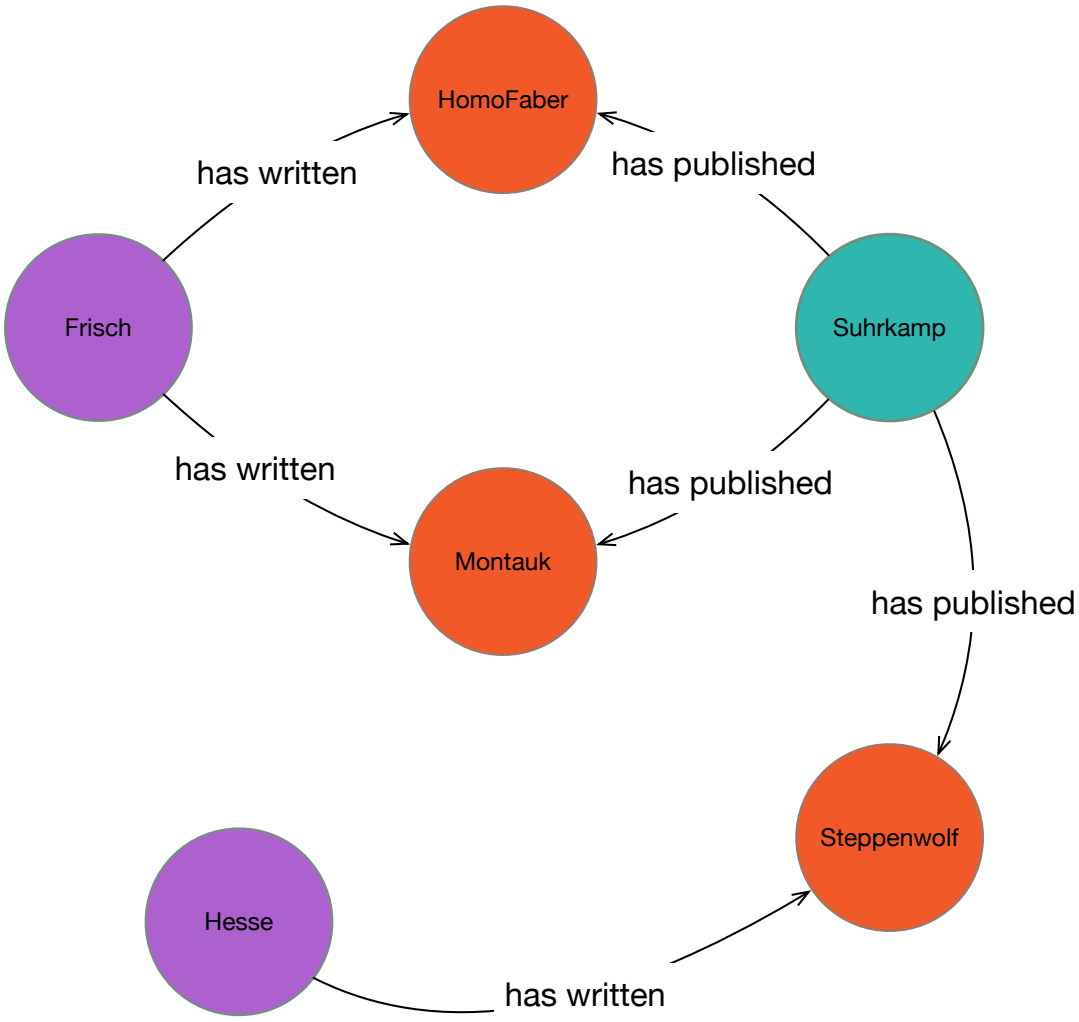



Figure 2: Simple sample Graph

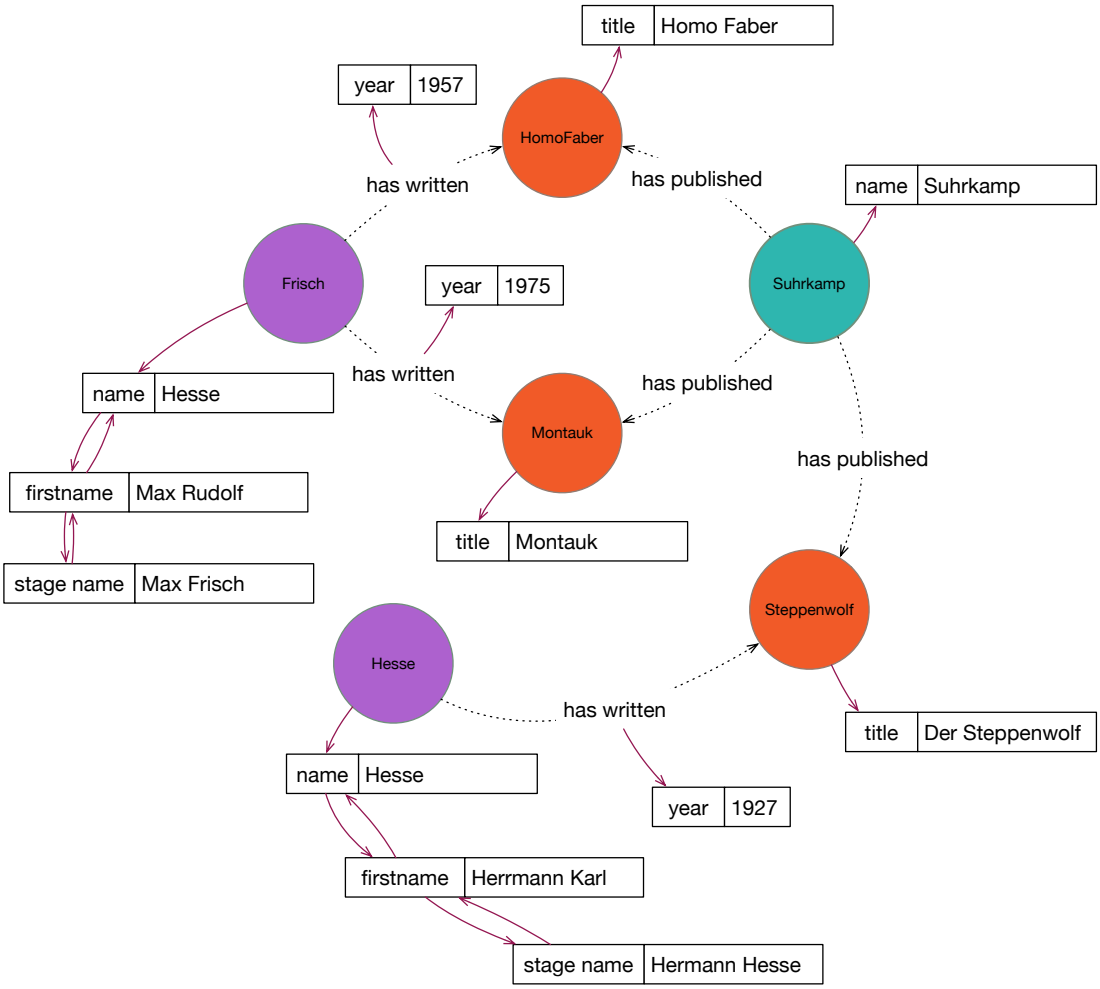



Figure 3: Property Records

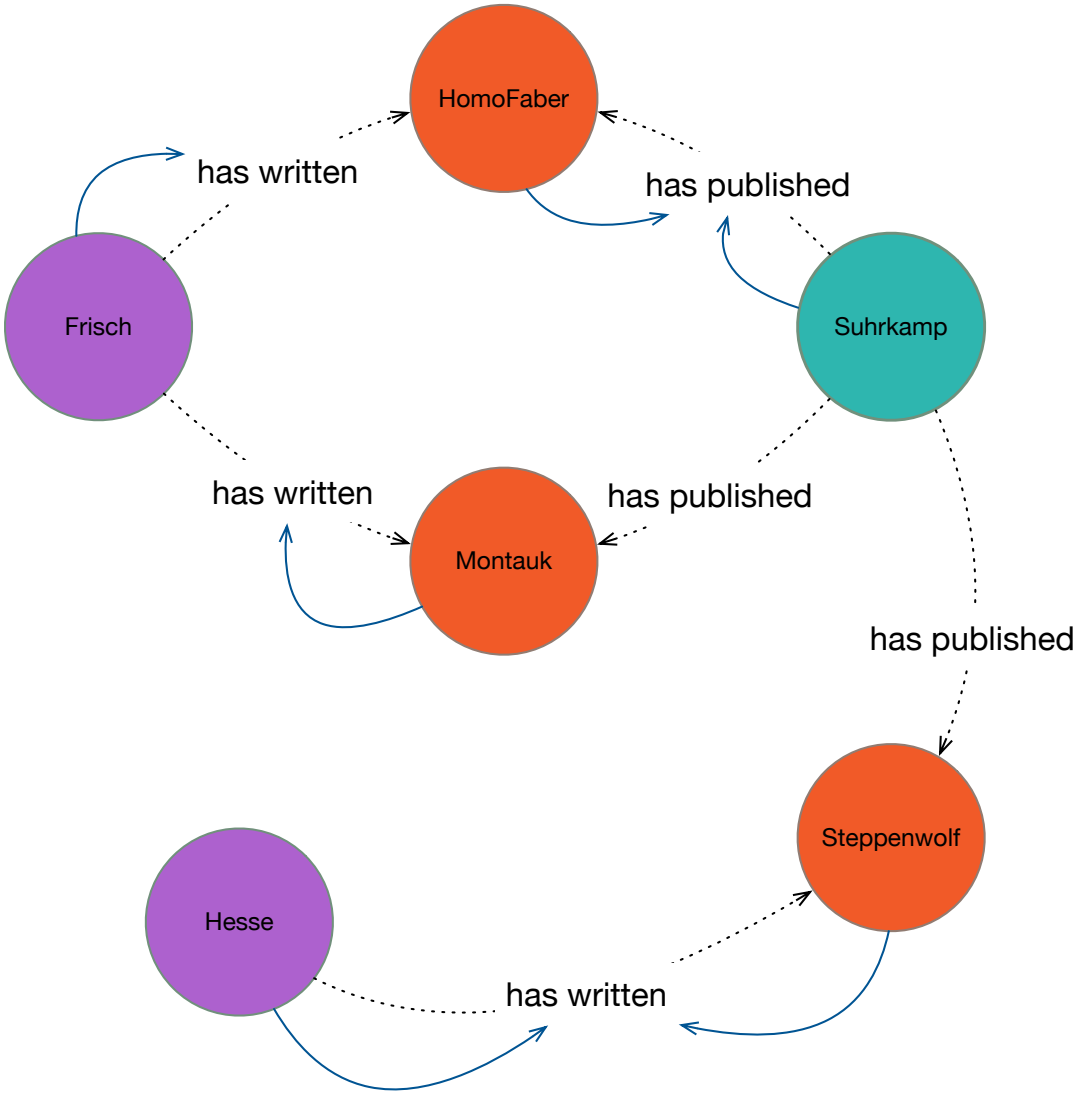



Figure 4: First Relationship

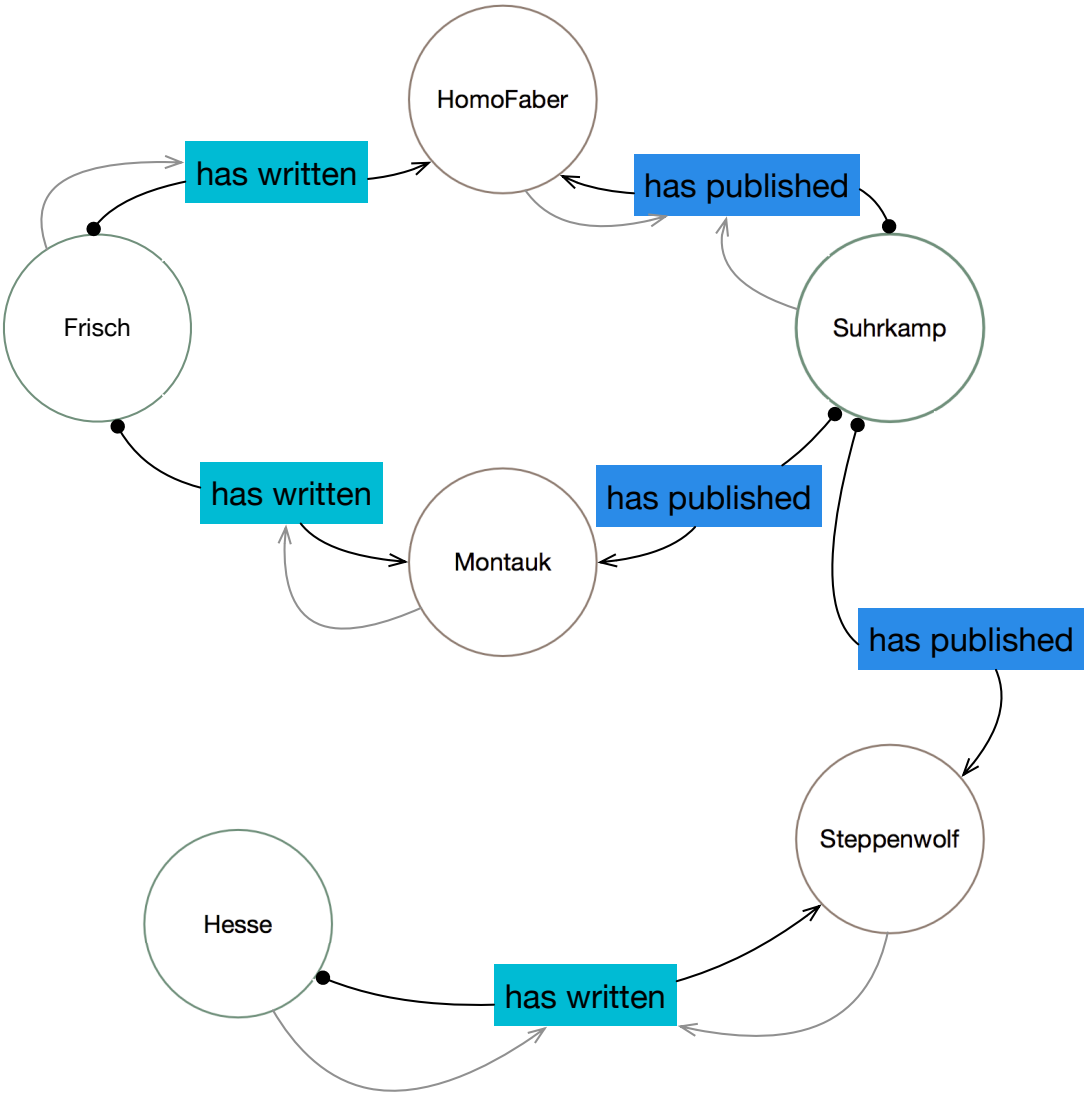



Figure 5: Start Node - End Node

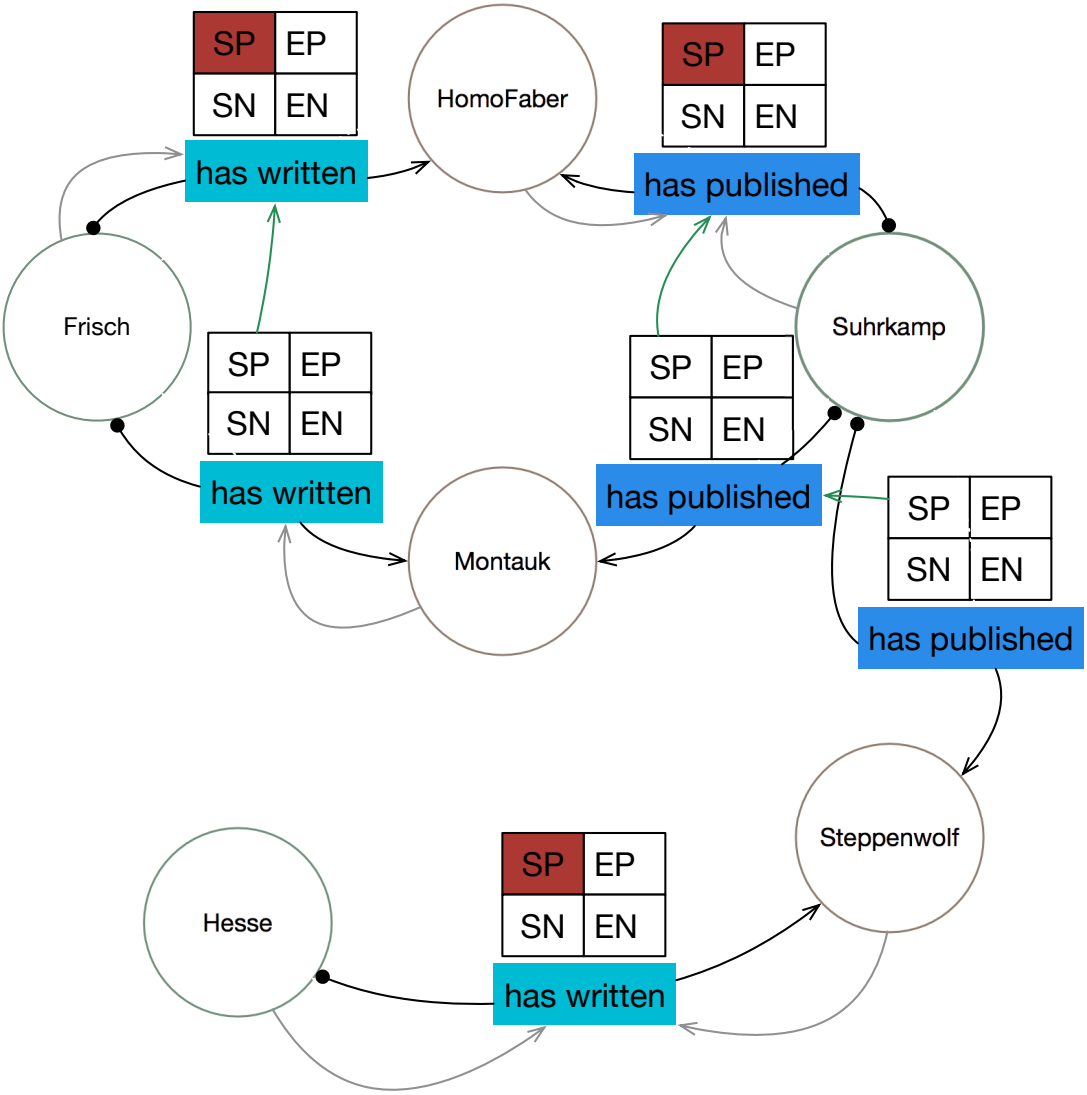



Figure 6: Start Node Previous Relationship

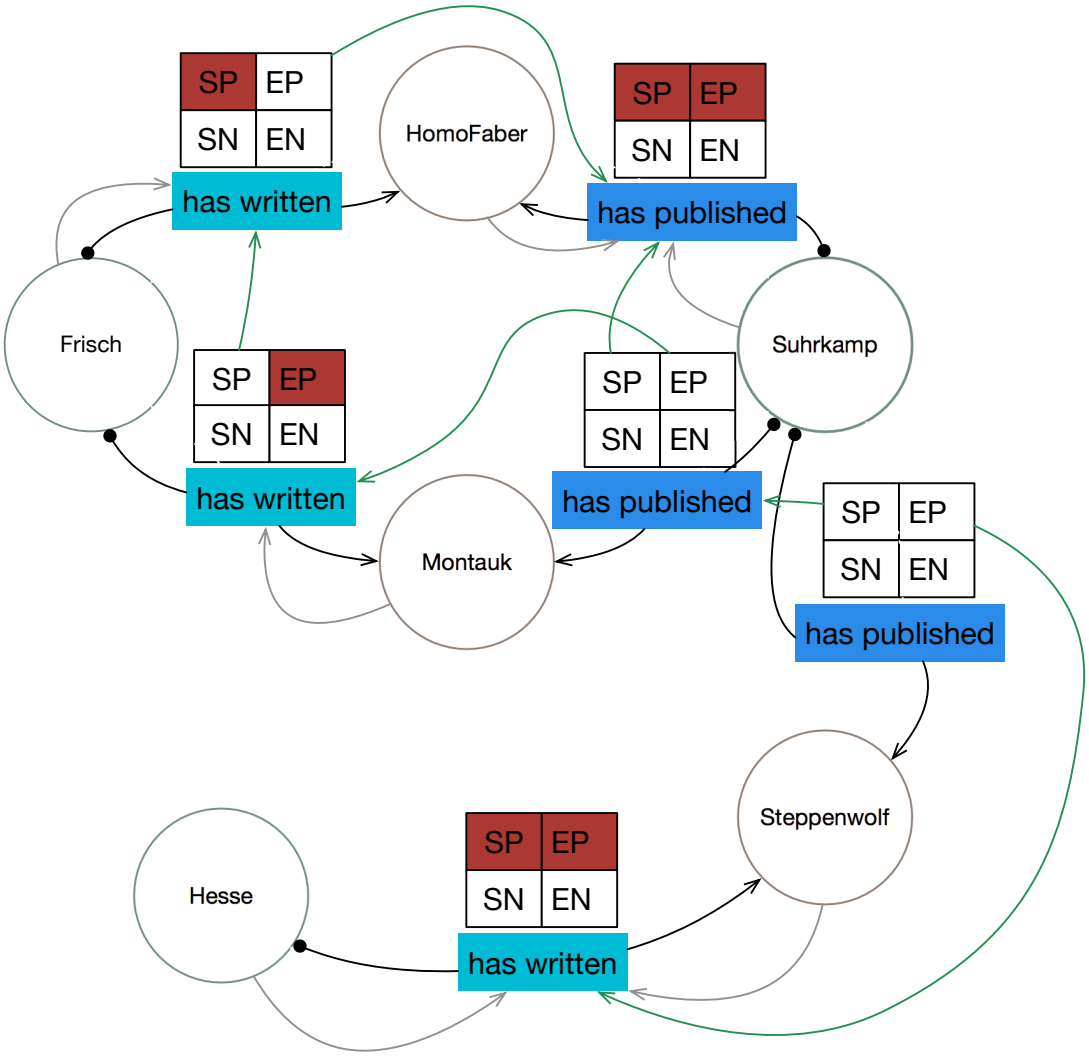



Figure 7: End Node Previous Relationship

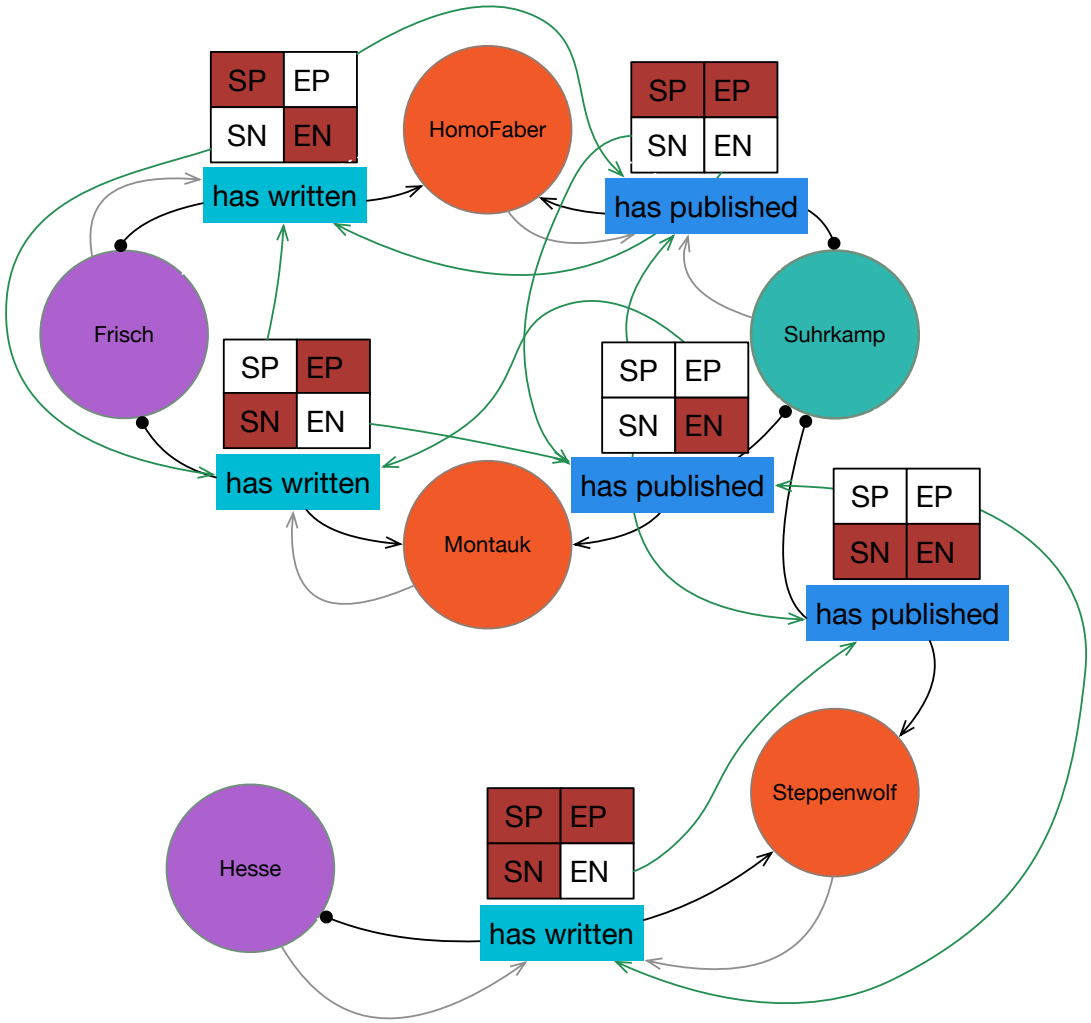
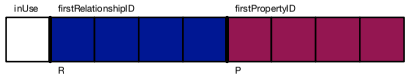
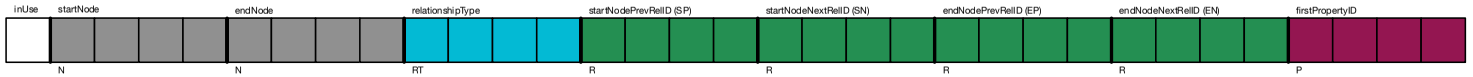



Figure 8: All Pointers Displyed

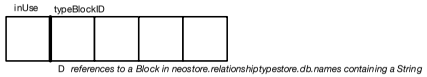
Node *N* (9 bytes) - *stored in neostore.nodestore.db*



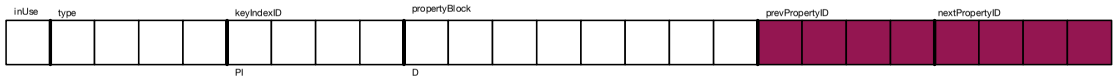

Relationship *R* (33 bytes) - *stored in neostore.relationshipstore.db*



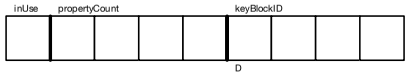

Relationship Type *RT* (5 bytes) - *stored in neostore.relationshiptypestore.db*



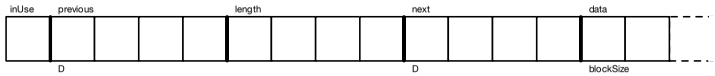

Property *P* (25 bytes)




Property Index *PI* (9 bytes)




Dynamic Store *D* (13+blockSize bytes)




NeoStore (5 bytes)

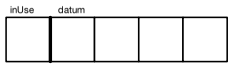



Figure 9: NeoBytes