Neo4j Graph Database

Nico-Alexei Hein, Lucas Thielmann, Andreas Rau & Lisa Mischer

31st March 2015

Abstract

This is a short explanation of Neo4j internals. This article gives some advice on how to work with Neo4j, including modeling and Business Intelligence.

1 Introduction

Neo4j is a transactional and graph based database. It was first released in 2010. Neo4j is written in Java and available under two different licences.

1.1 Neo4j DB

Neo4j has its own query language which is called Cypher. Cypher concentrates on pattern search. It is similar to SQL, both are descriptive languages. Cypher can be used in the command line, but all properties about transaction behaviour are also valid for transactions executed by a terminal or a program.
Queries are run in a transaction. Either there exists already a transaction or one is created. It is possible to commit several queries to one transaction. "A transaction will either fully succeed, or not succeed at all" Changes of a query are held in memory until the hole query has finished executing. This needs a lot of heap space and therefore it is important to structure queries. Neo4j supports the ACID properties - atomicity, consistency, isolation and durability. In case of a failure of a transaction a roll back is done. Neo4j offers several options for managing locking of resources. On the one hand there is a default locking behaviour, on the other hand tools are given to archive a higher level of isolation for transactions. Additionally, it uses deadlock detection algorithms.
Neo4j databases are accessible via a Java API and therefore can be embedded in Java applications. It also supports batch processing for large amounts of data.

1.2 Example

As an example we modelled the relation between books, authors and publishers. Authors write books and publishers publish books. To start working with the 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.
Figure 1 shows how our example looks in Neo4j. This picture is given by Neo4j in the browser view.

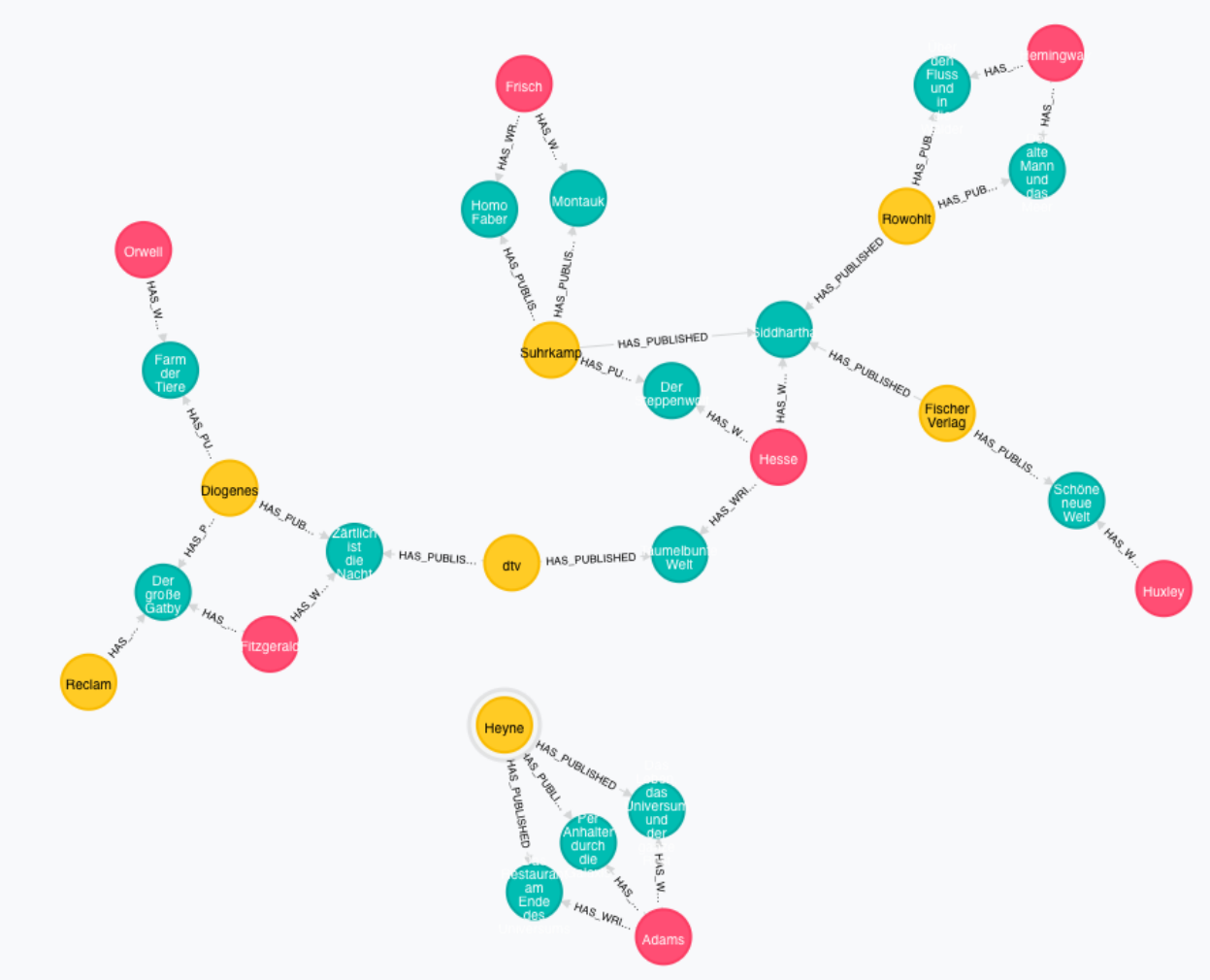

Figure 1: Example Graph

1.3 Queries

Neo4j offers a lot of possibilities for queries. In this section, some query examples are given in Cypher syntax for the example graph.
The syntax of Cypher statements are similar to SQL statements. Listing 1 shows the basic structure of a statement.

```
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. Listing 2 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 colon 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 braces. The node here has three attributes: name, first-name and stage-name.

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

Listing 2: Create Node

The statement for creation of a relation looks a bit different. Listing 3 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 brackets. Beginning with a colon the relationship type is defined, followed by all properties in braces. After the relationship information are finished an arrow points to the end node, again defined by the node ID in parenthesis. To use the ID to identify a node is only possible if the node is created in the same transaction as the relation. Otherwise the node has to be found via the MATCH statement. On the other hand, if one ore two nodes do not exist when create relation statement is executed, this nodes will be created automatically.

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

Listing 3: Create Relation

MATCH MATCH is used to find nodes. The simplest MATCH statement is shown in listing 4. This statement returns everything in the graph. E.G. this statement was used to

extract figure 1, an image of the graph which is shown above. Furthermore it is used for nearly every operation in the graph. One can find nodes of a special type, nodes or relations with a certain property or just certain information. If an item is found the statement can be extended, e.g. to add a CREATE statement or add a new property. Listing 5 is an examples which shows how to create a new relation between two existing nodes. This statement sets Heyne as the publisher of Homo Faber.

```
MATCH n RETURN n
```

Listing 4: Simplest Match Statement

```
MATCH (bb:Book), (vv:Publisher) WHERE bb.title = "Homo Faber" AND vv.name
    = "Heyne" CREATE (vv)-[r:HAS_PUBLISHED]->(bb) RETURN r
```

Listing 5: Match and Create Relation

Neo4j offers the possibility to solve a graph. It provides statements to find the shortest path through a graph (also possible for sub-graphs) or select the surrounding of a node with a certain radius. These statements are on the basis of the MATCH statement too.

DELETE In Neo4j it is only possible to delete the graph. This means, all nodes and relations are deleted, but all labels will remain. These can only be delete if the whole database is deleted. The DELETE statement also uses the MATCH statement as a basis. First, all nodes and all relations of these nodes are selected. It is important that a node which is to be deleted has no relations any more. It is not necessary to first to delete all relations and divide the operation in two transactions. With OPTIONAL MATCH this is done automatically by Neo4j. The DELETE statement takes every selected item and deletes it.

```
MATCH (vv:Publisher), (aa:Author), (bb:Book) OPTIONAL MATCH (vv)-[hp]-(),
    (aa)-[hw]-() DELETE aa, vv, bb, hp, hw
```

Listing 6: Delete Graph

2 Data Modeling - Describing a Domain

Data models in Neo4j are not enforced strictly. Instead the designer has to ensure a consistent database by only adding data that conforms to the desired model. The following paragraphs will describe step by step how such a model can be created.

2.1 Nodes

The process starts with identifying all nodes that exist in your domain. A node is any unique physical or abstract object that is important in the context of the domain.

Examples:

- theGreatGatsby
- fScottFitzgerald
- reclam

2.2 Labels

In the next step, all labels of the domain have to be identified. Labels assign a specific role to a node. Nodes that have the same label are grouped into a set. These sets can later be used to write more efficient queries by only considering small subsets of a graph.

Examples:

Label	**Node**
Book	theGreatGatsby
Author	fScottFitzgerald
Publisher	reclam

2.3 Relationships

To transform the collection of labeled nodes into a graph, the next step identifies the relationships between the nodes. A relationship is a directed arc from one node to another. This relationship has an identifier that can later be used to query connected nodes.

Examples:

relationship identifier	**From Node**	**To Node**
HAS_WRITTEN	fScottFitzgerald	theGreatGatsby
HAS_PUBLISHED	reclam	theGreatGatsby

2.4 Data

In the last step, the graph is populated with additional properties. Data can be attached to nodes as well as relationships. These properties are often the information that provide the end result of a query. They can also be used to narrow down result sets and filter for appropriate results.

	Property	**Node/Relationship**
	title:"The Great Gatsby"	theGreatGatsby
Examples:	firstname:"Francis Scott Key"	fScottFitzgerald
	name:"Reclam"	reclam
	year:"1925"	HAS_WRITTEN

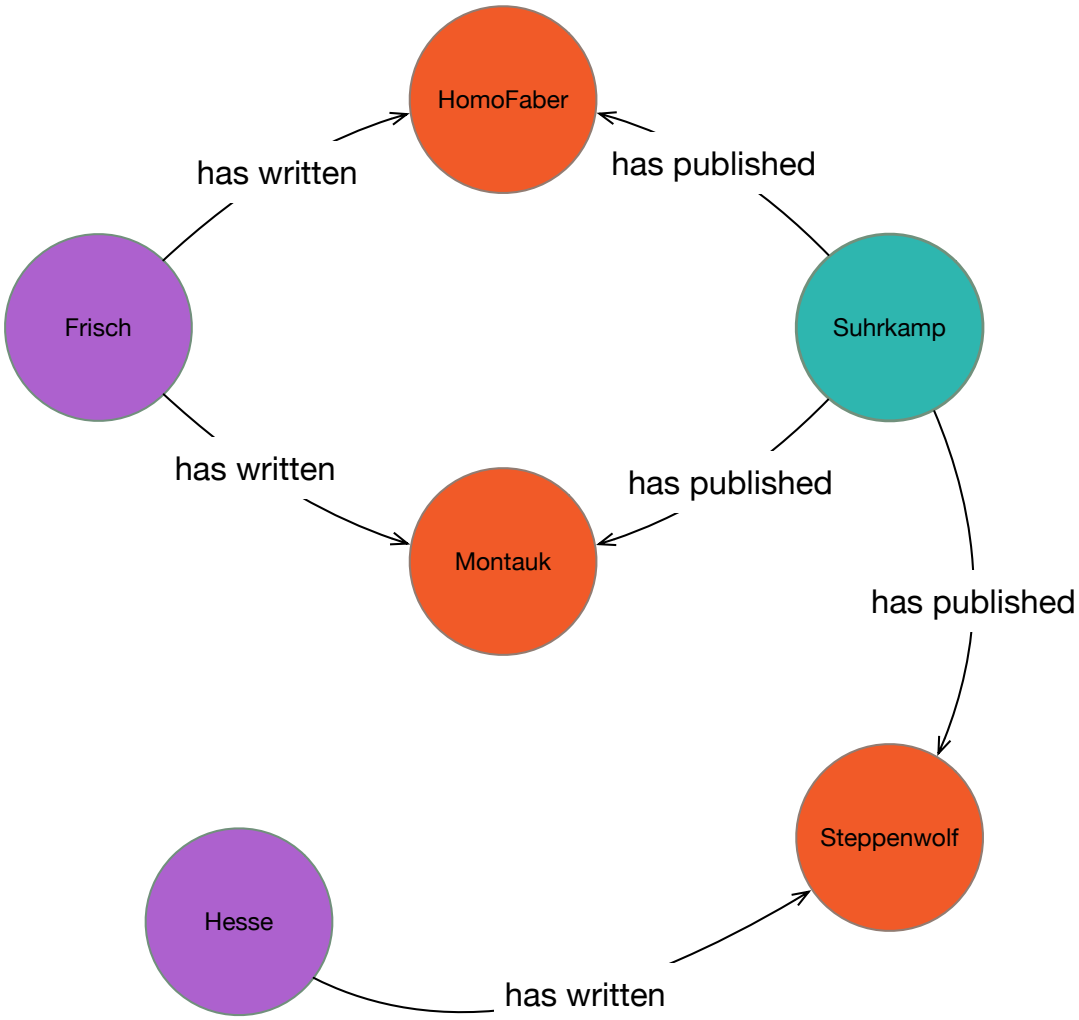

Figure 2: Simple sample Graph

3 Storage Structure

3.1 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.

3.2 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 at the moment). To store the

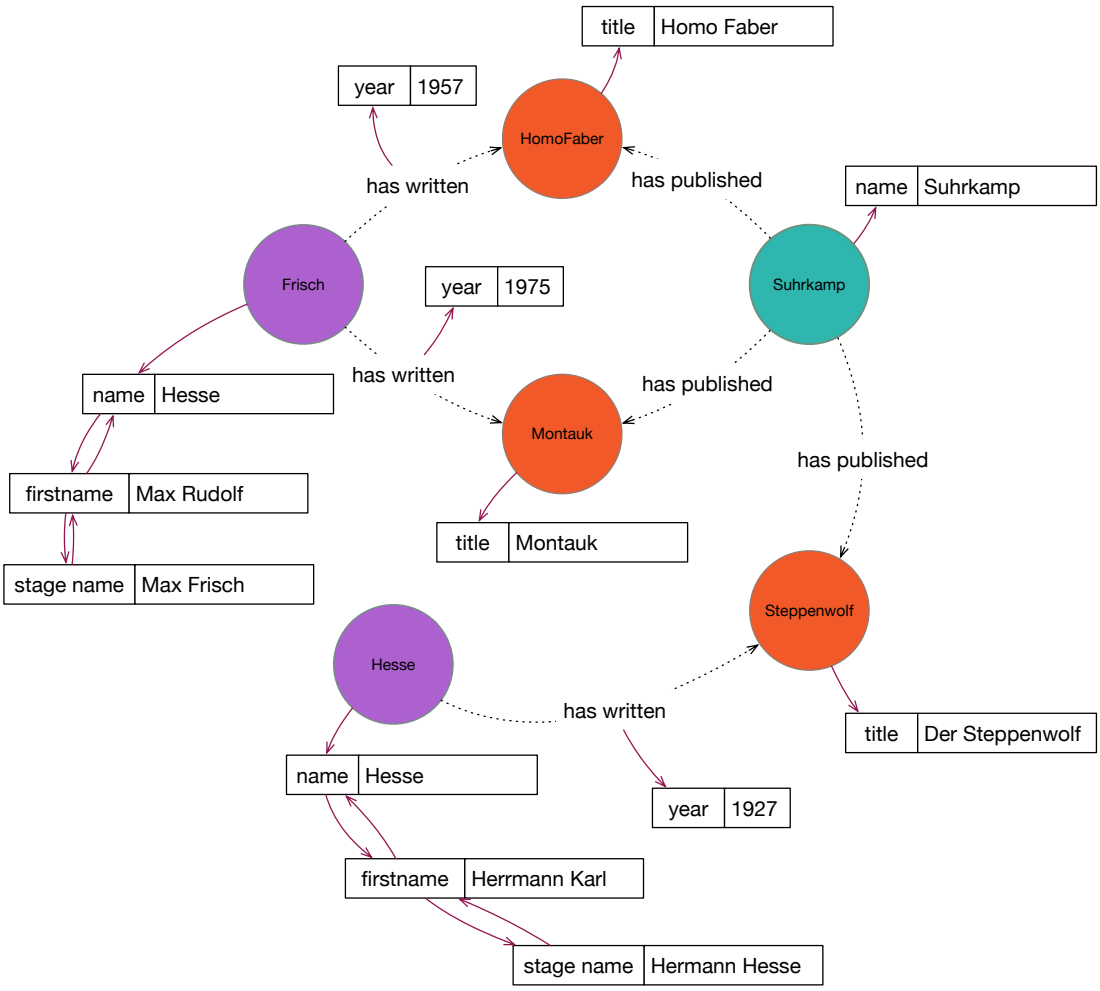

Figure 3: Property Records

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.

3.3 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.

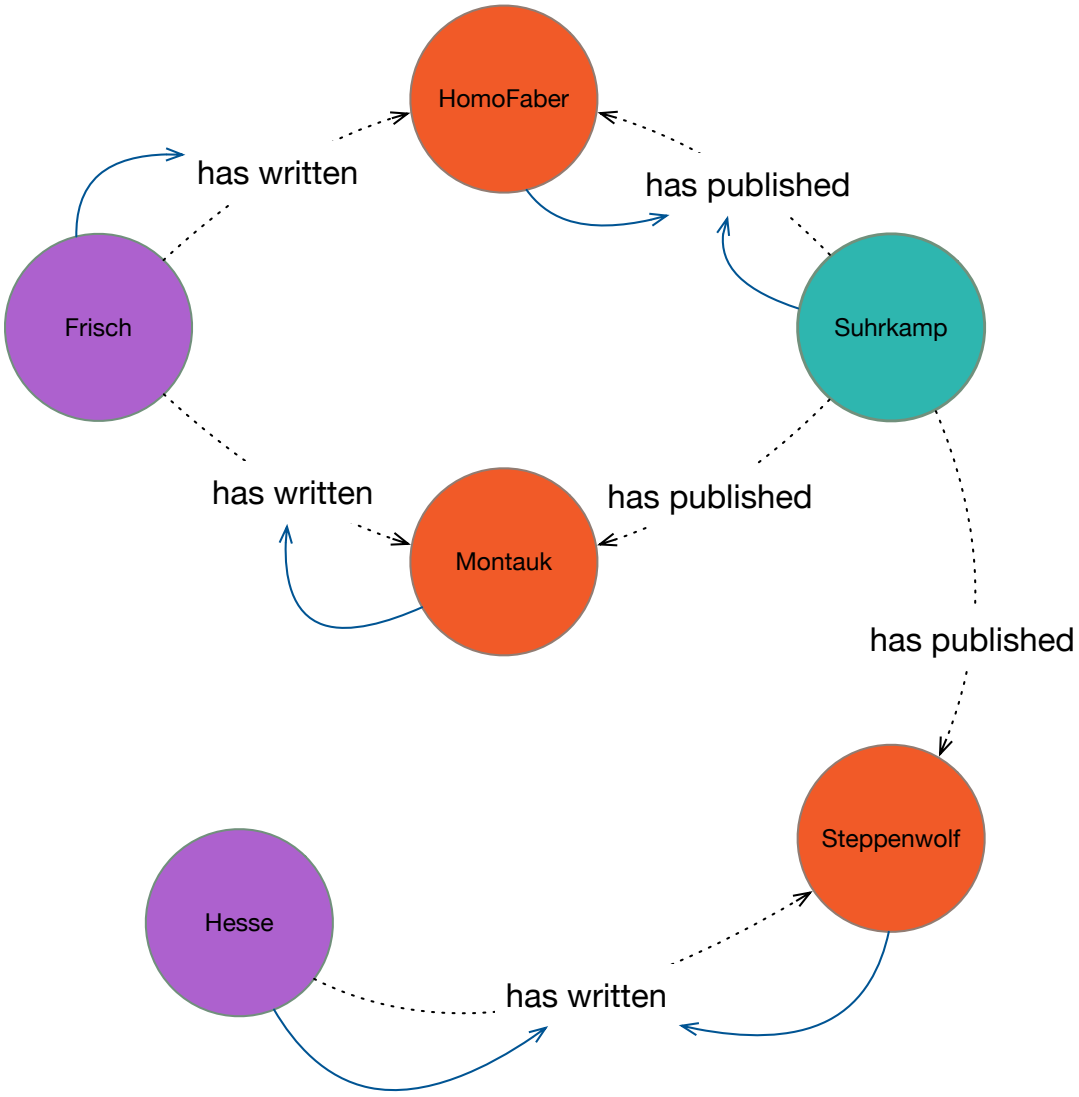

Figure 4: First Relationship

3.4 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.

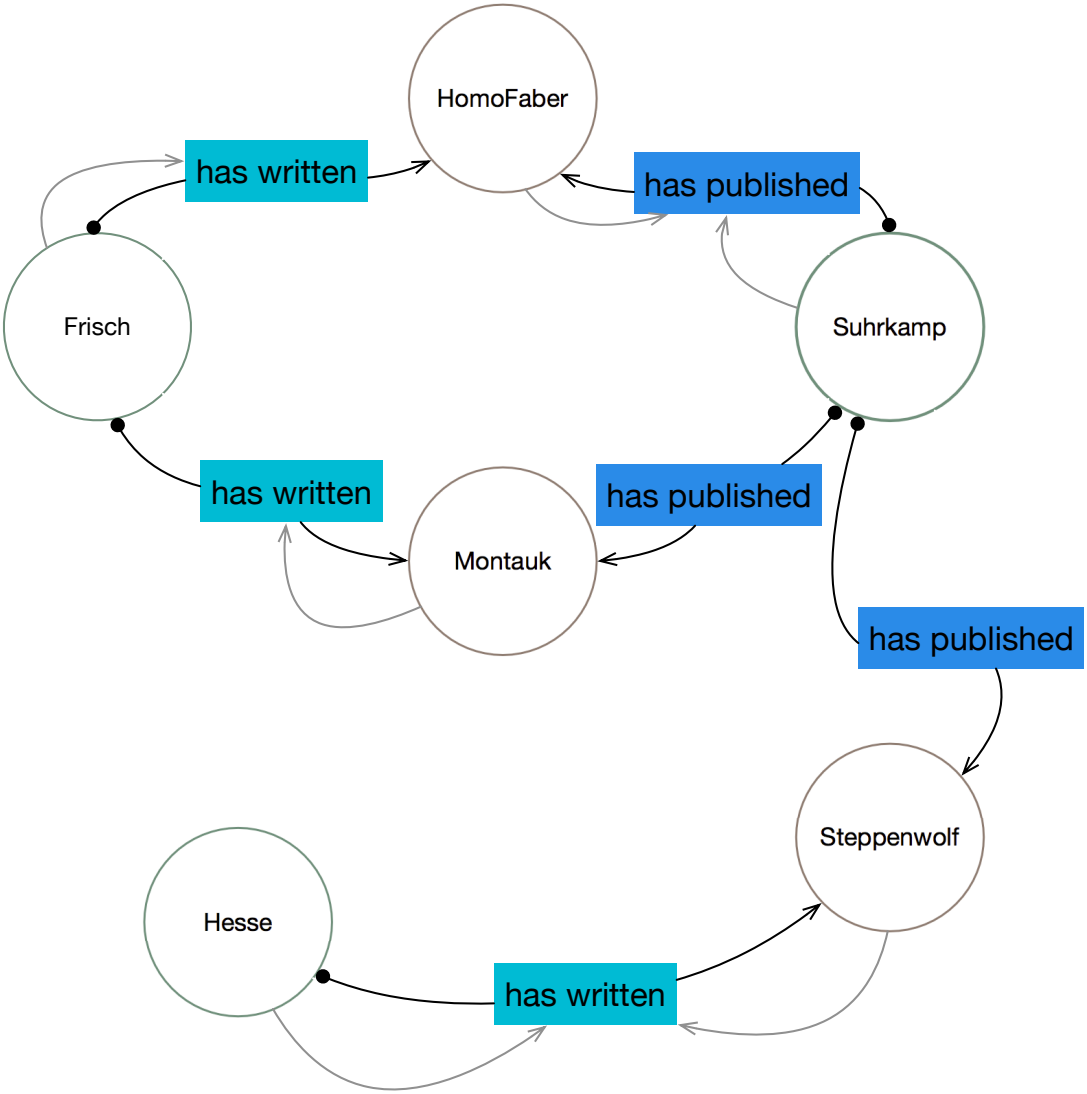

Figure 5: Start Node - End Node

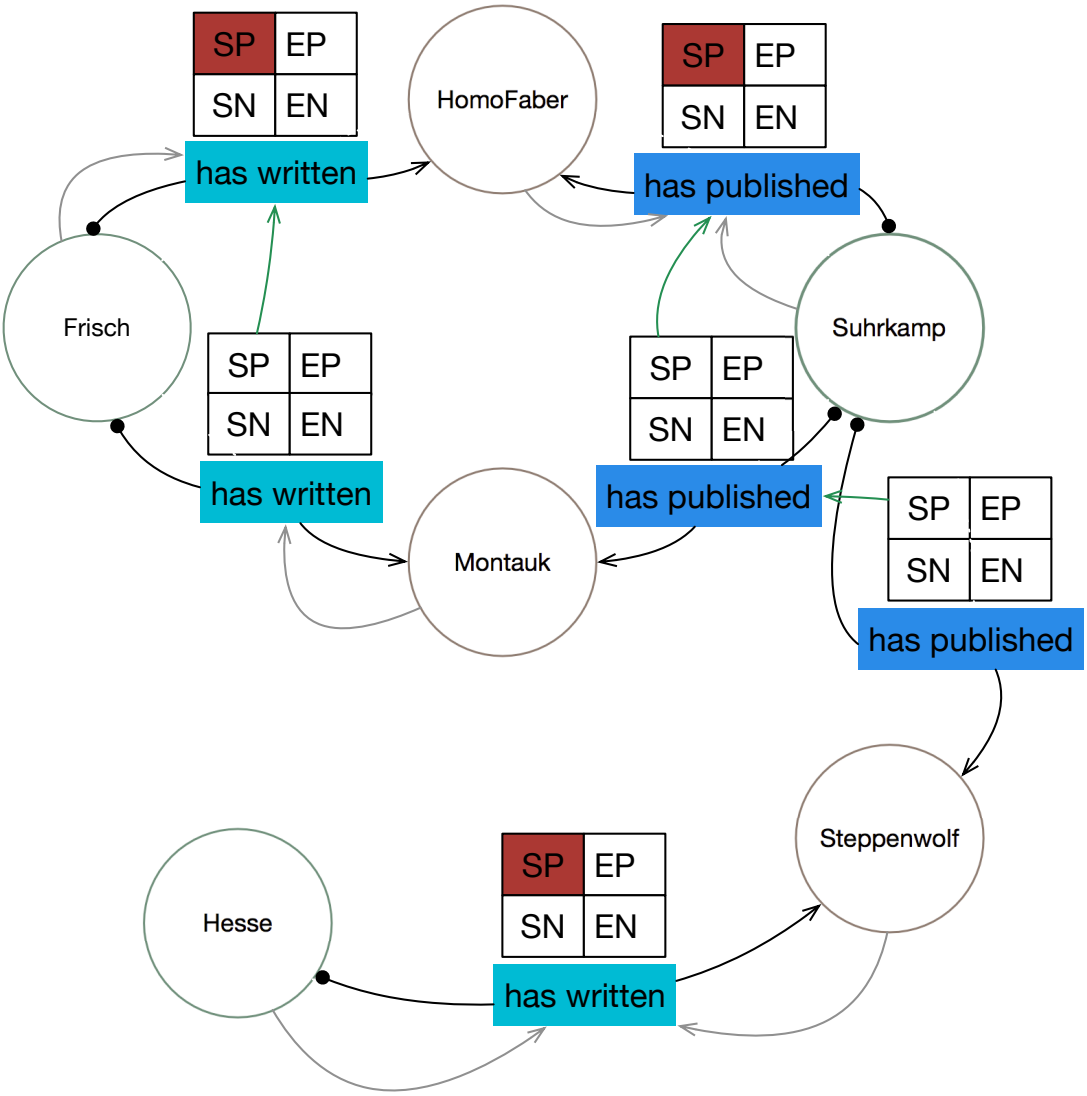

Figure 6: Start Node Previous Relationship

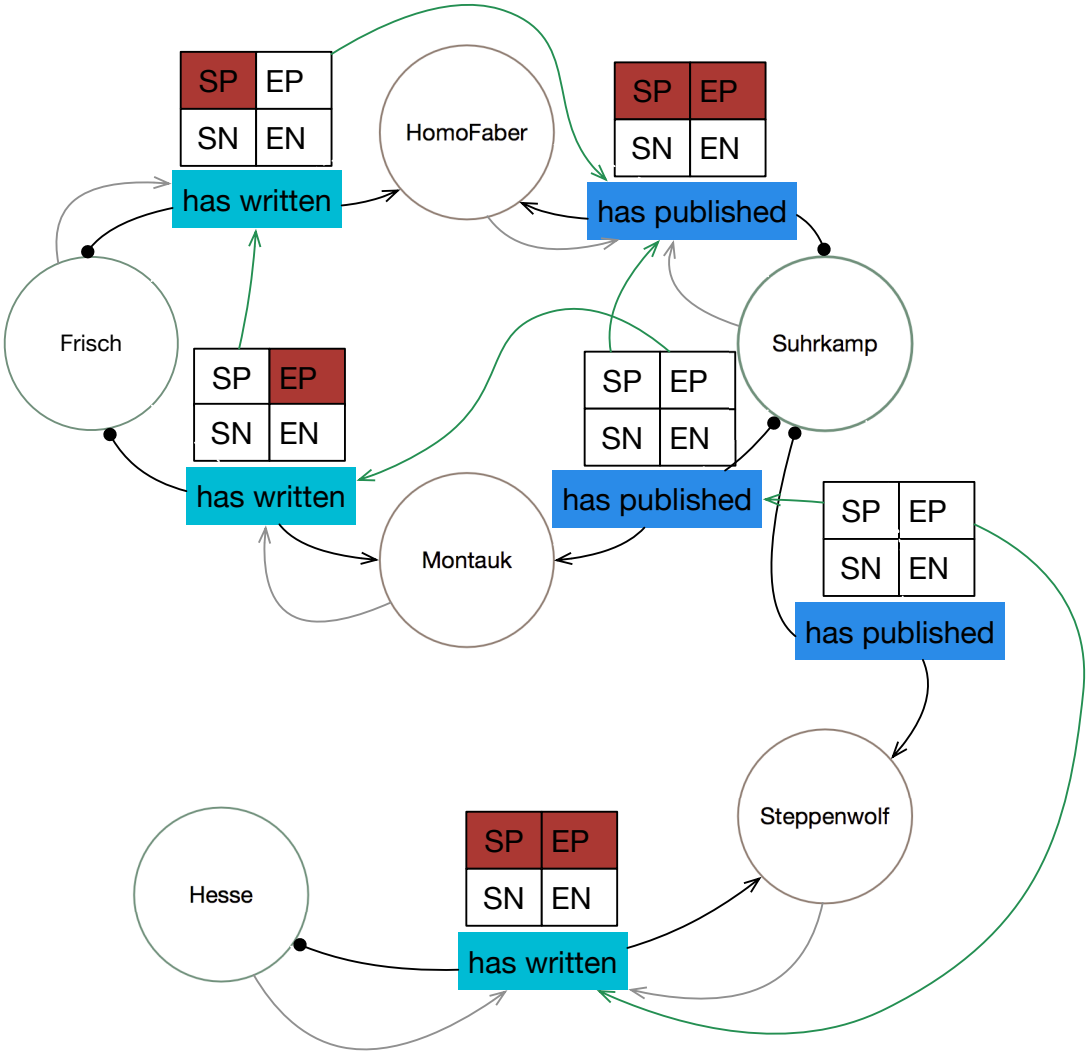

Figure 7: End Node Previous Relationship

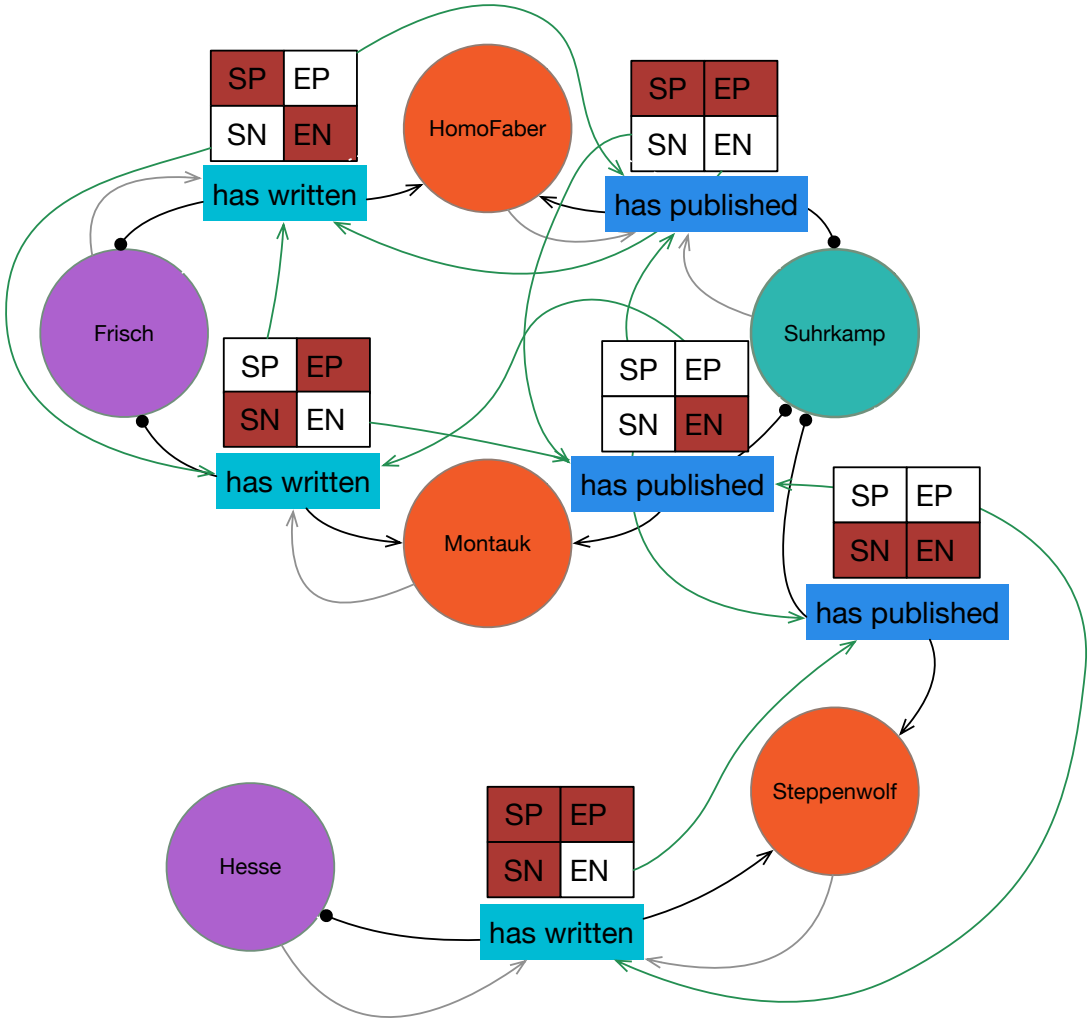

Figure 8: All Pointers Displyed

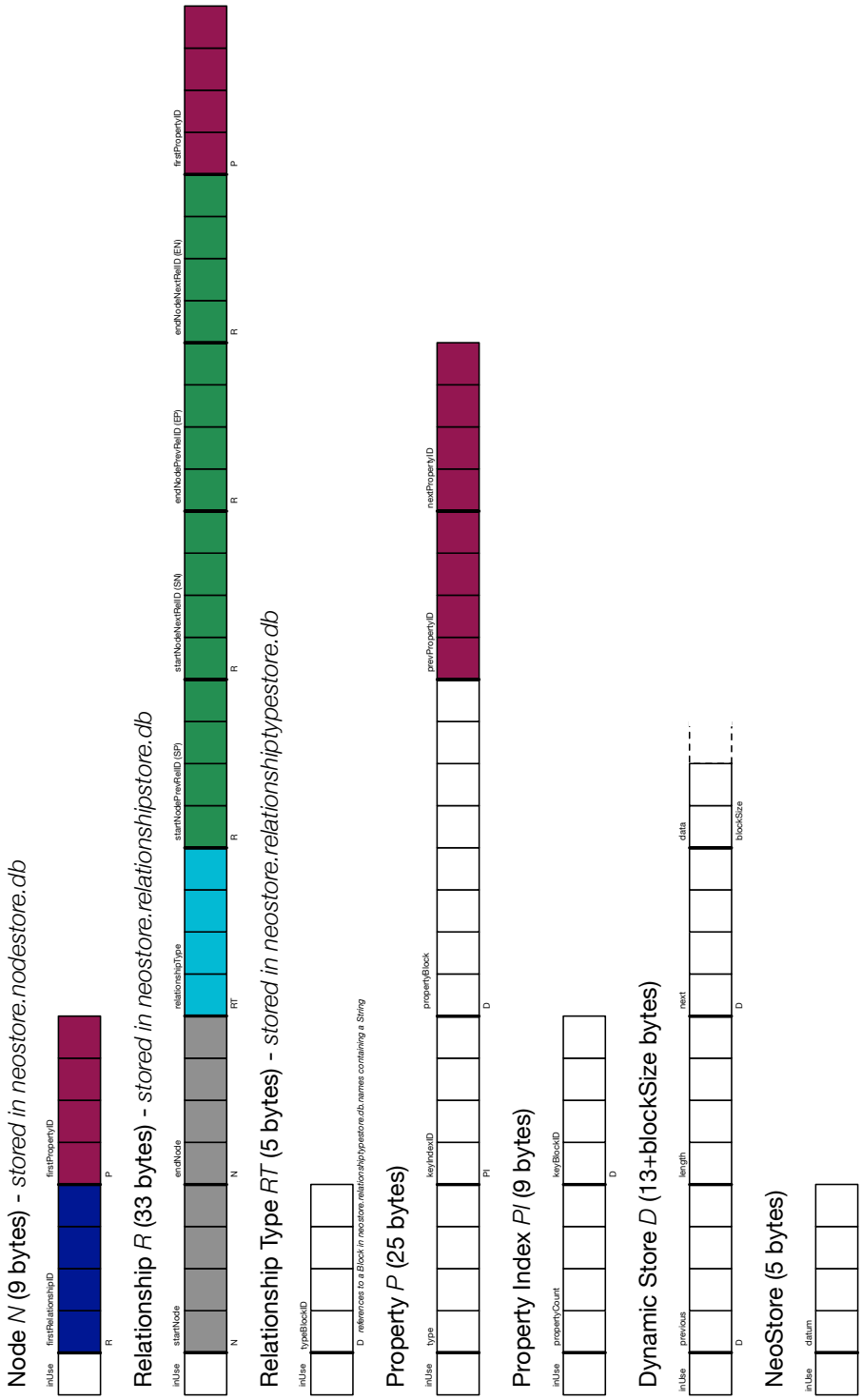

Figure 9: NeoBytes

4 Neo4j in BI

4.1 Business Intelligence

"Business intelligence (BI) is a term that refers to ideas, practices and technologies for turning raw data into information that businesses can use to make better organizational decisions. Businesses that employ Bi effectively can transform information into growth by gaining a clear understanding of their strength and weaknesses, cutting costs and losses, streamlining internal processes and increasing revenue." **neo4jBIDef:2014**

4.2 Why are graph databases good for BI?

Graphs can easily visualize data based on how it exists in reality. This is shown in our book example. There are no complexe transformations needed in order to store information. The more data a company gathers the more complex it gets to model the information in order to use it. This is the cause of the fact that data usually is inherently hierarchical and/or connected. BI relates on the useage of large amounts of data but still misses performance when it comes to "transform information into growth" **neo4jBIDef:2014** Business demands are increasing with the amount of customers and therefore the amount of data and because of the usual deep diving hierarchical structures of regular SQL databases the results are long searching times as well as complexe search queries. As we all know IT is a real time business. No bank transfer can wait minutes or seconds for its closure. User expectations are growing and there have to be new alternatives to store data. Graph databases are an entrenched solution to this problem, Neo4j is the market standard and a few steps ahead of others because it stores data in a real graph.

4.3 BI and graph databases

Graph databases can, like any other type of database be used in many applications of interest. One of them is Business Intelligence. With the help of analytical queries data can be "naturally expressed via an OLAP-like aggregation framework" Dritan Belco and Yannis Kotidis, 2013. By using simple graph theory like weighted edges, productions lines as well as the shipment of products from the producer to the customer can easily be mapped into a graph database. From there on factors like location, storing, capacity etc. can be analyzed for instance to minimize waiting times of the product at different stages of the delivery or to spot the fastest and slowest routes from one location to another. Because of the size of

data grouping and aggregation is required in order to pre-computate information within the database in form of materialized views containing frequently asked aggregates. This pre-processing enables the use of datawarehouses and more analytical processing on the data, also with the regard on more dimension ‘Neo4J Enterprise vs. VoltDB’, n.d.

References

Chris Gioran. (2010). Neo4j internals: file storage. Retrieved from http://digitalstain.blogspot.de/2010/10/neo4j-internals-file-storage.html

Dritan Belco and Yannis Kotidis. (2013). Business intelligence on complex graph data. Retrieved from http://www.edbt.org/Proceedings/2012-Berlin/papers/workshops/beweb2012/a3-bleco.pdf

Huston Hedinger. (2013). Business intelligence and analytics with the graph. Retrieved from http://de.slideshare.net/neo4j/0905-25970888

OECD. (2011). The neo4j manual v2.1.7. Retrieved from http://neo4j.com/docs/stable/query-transactions.html

Neo4J Enterprise vs. VoltDB. (nodate). Retrieved from http://vschart.com/compare/neo4j/vs/voltdb

Tobias Lindaaker. (2012). An overview of neo4j internals. Retrieved from http://slideshare.net/thobe/an-overview-of-neo4j-internals