Graph databases by the example of Neo4j

# Introduction

A great deal of today’s software applications are built around databases and perform various tasks on large data sets to serve certain business interests. Relational databases have been around for decades and have been optimized to serve this purpose best. However, with the growing of the internet and the vastly increasing amount of data (big data), new types of databases have emerged under the designation NoSQL. Common representatives of this kin are key-value stores and document-oriented stores, but also the less established graph databases. In contrast to its relatives, the latter focuses strongly on relations and the topology between entities instead of the entities itself. Graph databases therefore provide customized query languages with rich expressiveness regarding relations and also feature algorithms from graph theory. They are therefore suitable for problems where efficient graph traversal and related graph algorithms are essential.

The subject of this document is to examine the usability, performance and possible application scenarios of graph databases. As an example for a graph-related problem, two games based on the Wikipedia will be demonstrated:

1. Wikipedia Philosophy game  
   Select a random article. Follow the first non-italic, non-parenthesis-enclosed link of the main text and repeat this step until a cycle is detected or the article Philosophy is reached.  
   Several persons analyzed this issue und came to the conclusion that roughly 95% of all articles link to Philosophy this way[[1]](#footnote-1).
2. 5 clicks to Jesus

Select a random article. Using all links of the main text, try to reach the article Jesus in 5 clicks (links) or less.

# Data sets

Three dumps of the Wikipedia articles have been used throughout the tests:

|  |  |  |  |
| --- | --- | --- | --- |
| Language | Date of dump | # Articles | # Links |
| English | 2014-12-08 | 15113788 | 164379808 |
| Russian | 2015-01-10 | 3039356 | 36903313 |
| Plattdütsch | 2015-01-08 | 607331 | 31814 |

The English Wikipedia was used initially with the goal to solve the original philosophy game. However, during preprocessing we came to the conclusion that the data set was far too large to build a fair database for benchmarks and tests. The reason for this is that, although all articles have ids, Wikipedia stores its links using the textual name of the linked page, and not its id. Still, well-designed relational databases use integral ids to refer to foreign entities. Considering a fair comparison, we tried to resolve all textual links (several million) to their page ids. However, with <10 resolved ids per second due to the large dataset, we aborted this process as it would have probably taken months to complete. It would be possible to resolve the ids without a database using e.g. a custom coded program and a hash map with enough size to store 10GiB strings as keys with associated values. Nevertheless, this would not solve the equivalent problem when inserting into a graph data base where relationships can only be created after still matching the destination page by either string or integer. The ultimate choice was to drop the large data set and try a smaller one. First experiments with the Russian Wikipedia showed that despite all indexes resolving all pages would still require approximately 11 days. Furthermore, due to the language’s nature, Unicode support became an essential requirement, doubling the storage size of the database. While inserting all links, we discovered that our SQL database (LocalDb) has a limit of 10 GiB per table. Thus also the Russian Wikipedia was too large for our experiment. Finally, we have chosen a quite small Wikipedia which can still somehow be understood, the Plattdüütsch Wikipedia.

# Handling large files

The database creation with the selected test data was preceded by a long period of preprocessing (several weeks). It turned out, that the size of the original English Wikipedia dump (~ 49 GiB) was quite hard to handle. Here is a list of difficulties we had to face during processing the dumps and how we got around them:

* The file size of ~ 49 GiB  
  Files with such sizes cannot be opened in ordinary text editors like Notepad++. We tried a number of proprietary (shareware) text editors claiming to be able to deal with this dimensions. Although a lot of tools could view the file, none was capable of performing operations like search&replace or regular expressions at reasonable speeds (e.g. search&replace at 27 lines/s on a 164.379.808 lines file). We therefore used a tool to split the first e.g. 10 MiB form the file to analyze the data layout and try some transformations. If they worked out, we coded them using C# and processed the file line-by-line.
* Lines longer than String can hold  
  Initially, all destination links of a page where parsed from the wiki markup of each page. We later found out that Wikipedia also provides SQL dumps of their link table which is generated from the wiki markup. However, this SQL dump contained all values as tuples of a single line INSERT statement. With several million tuples on a single line this data could only be read chunk-by-chunk which is impractical when we want to run a regular expression over multiple chunks. We therefore dropped the SQL dump and stayed with the links parsed from the wiki markup.
* String.Split() fails to allocate memory  
  The implementation of String.Split() inside the .NET framework returns an array of all substrings split from the original one. From reading through the source code we also found out that quite some temporal data is also allocated during splitting. This caused String.Split() to fail when splitting larger lines approaching e.g. half a GiB. We therefore implemented a lazy version of splitting strings.
* Regex and performance  
  Link extraction from wiki markup is done using regular expressions which run on the wiki markup of each page. During a performance analysis we found out that most of the time our preprocessing tools were not bound by disk IO but by the execution speed of the regular expression. Therefore, processing of pages and link extraction has been parallelized and compiled regular expressions where used to process dumps at reasonable speeds (e.g. 40 min compared to several hours when run on the English Wikipedia)
* SQL INSERT throughput  
  Initially, our preprocessor generated SQL scripts with millions of SQL INSERT statements. Despite connection pooling and even holding a single connection for all queries, the database could only process around 1000 queries per second which was far too slow for a dataset with several million records. Therefore BULK INSERTs where used with appropriately generated CSV files yielding a much higher throughput
* Unicode  
  By switching to a non-English Wikipedia, support for a wide range of characters became a requirement. Switching to Unicode seemed a good option but introduced a bunch of problems. First, Unicode allows a single textual character to be represented by multiple physical characters (e.g. combining characters and surrogates). Splitting a string or creating a substring was suddenly a non-trivial task. Furthermore, SQL databases require special column types to support Unicode (e.g. nvarchar). Finally, also the various encodings were a little annoying as MS SQL Server (and LocalDb) does not support UTF-8 whereas our graph database of choice (Neo4j) requires input files in UTF-8.

# Method

# Input preprocessing

# MS SQL Database (LocalDb)

# Neo4j Database

# Benchmarked queries

Insert script took (eng)

00:46:00.1983008 (b)

00:41:19.1609778 (p)

# Conclusion

1. http://matpalm.com/blog/2011/08/13/wikipedia-philosophy/ [↑](#footnote-ref-1)