Lowering triple store latency through application object caching: optimisation of a caching algorithm

Rosanne Vreugdenhil, 4163869
Project Supervisor: Prof. dr. Toine Pieters
Second examiner: I. Troost, MSc

MSc Applied Data Science

Graduate School of Natural Sciences
Utrecht University
02-07-2021

1 Introduction

History is everywhere. With just a random stroll around Utrecht, one can come across varying historical buildings. In fact, the streets walked on in that stroll themselves carry a rich history as well. A lot of this history is freely available, but not always easily accessible. Utrecht Time Machine is part of the European Time Machine initiative and has the goal of bringing the history of Utrecht digitally to life by disseminating historical data sets in an intuitive and narrative-based way. By doing so, they are making the history of Utrecht more easily accessible and closer to everyone interested.

UTM collaborates with many partners, including the Utrecht Archives. Their lab (HUALAB) created a pilot called "Op de Kaart" (on the map) for geocoding the image collection from the archive. This application provides three maps: a map of all streets in the municipality Utrecht, a map of streets in the Netherlands outside of Utrecht that matched with the data of the archive, and a map of all train stations in the Netherlands. Figure 1 shows the first map of all the streets in the municipality Utrecht.

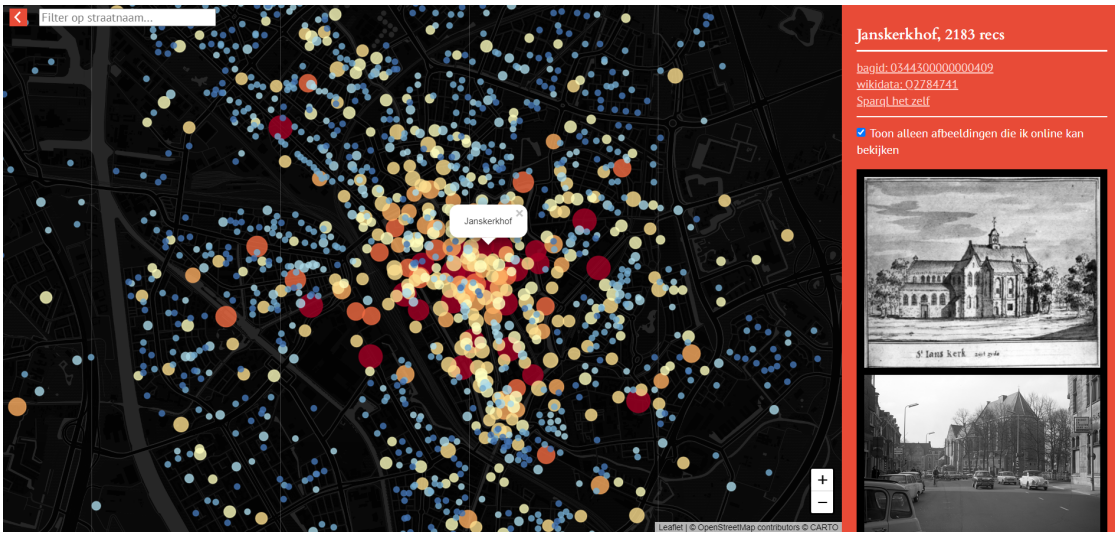

Figure 1: Op de Kaart - Streets of Utrecht

Clicking on a street on the map provides data for that street in Linked Open Data (LOD) format. Linked Data is using typed links between data from different sources, resulting in a web of things in the world. Berners-Lee (2006) outlined a set of rules called the "Linked Data Principles" to ensure that published data on the web all becomes part of a global data space. LOD specifically indicates Linked Data that is openly accessible. When a street name is clicked the application retrieves the corresponding data for that specific street from the archive, loading it into the site. All the images are accompanied by a description, a reference to where the image can be found in the archive, a license, and a date. The latter provides an opportunity to further organize the data disseminated by this application and add to its user accessibility. Right now, the user can search the map for street names. Adding the option to search for specific years or periods is a valuable addition for both the interested viewer

curious of Utrecht in a specific period, as well as a historian researching a period in the history of Utrecht. Horak (2016) emphasizes the usefulness of presenting historical data on digital maps in her overview of what film scholars and other spatial humanities projects have done with digital mapping tools. An example of this usefulness is the historical film study done by Verhoeven, Bowles, and Arrowsmith (2009) in which they discovered that Greek cinemas appeared in particular neighborhoods in Australia before an influx of Greek immigrants by mapping the mid-century Greek cinema circuit. Moreover, enriching the data accessibility and searchability by organizing it in periods could further add to the intuitive way the data is disseminated, contributing to the goal of UTM in disseminating historical data in an intuitive way and narrative-based way. A narrative element is added to the visualisation by assigning the locations a starting and ending event (earliest and latest data point), therefore indicating a chronological order in the data and emphasizing the temporal aspect of the images the user sees (Abbott, 2020).

Adding the option to search the data visualized by Op de Kaart by years will imply a change in the visualisation created by the application. The visualisation of Op de Kaart currently consists of a map with coloured circles for every street, in which the size of the circles indicates the number of data points available for that location. As the number of sources can vary year by year, the sizes of these circles will vary for each year when organizing the data in a temporal way. This way, each year will give a different visualisation on the map of Utrecht. The amount of data would be too large to generate these maps on-site when a specific year is requested without a high latency. This is a common problem for web applications based on triple stores (a storage of triples, required for linked data (Rusher, n.d.)), as the flexibility to amend and reorganize schema structures and the complex query constructs make them significantly slower than relational databases (Lorey and Naumann, 2013; Martin, Unbehauen, and Auer, 2010; Zhang et al., 2016). One often applied solution for this is caching application objects or a part of the generated user interface to lower latency (Martin, Unbehauen, and Auer, 2010). Caching is storing frequently accessed data in a temporary storage area so it can be accessed quickly (Gadkari, 2013). Martin, Unbehauen, and Auer (2010) evaluated the effect of caching query results and complete application objects on the performance of triple stores and concluded that their approach outperformed cacheless triple stores by more than factor 10.

To create the application cache for Op de Kaart, an algorithm is needed that is performant enough to handle a large amount of data while being executed in a short time so it can be executed regularly to prevent an outdated cache. An important aspect of this algorithm is its speed as the earlier mentioned high latency of querying on triple stores will slow down the caching algorithm in its turn as well. Optimisation of the algorithm will be needed to ensure that the execution time will be short enough to be executed regularly.

Another important aspect of this algorithm is its scalability: whether the algorithm can maintain the same efficiency when the workload increases (Bertossi, Pinotti, and Gupta, 2010; Bondi, 2000; Teng, 2016). Right now, Op de Kaart almost exclusively displays the streets in the city of Utrecht, with the exception of streets and train stations outside of Utrecht that are matched with data from the Utrecht Archives. This application could be expanded by adding more data from the Utrecht Archives or including more cities and their data in the visualisation by collaborating with other archives. In this case, the caching algorithm

should be scalable to larger numbers of data points while still being able to be executed in a reasonable time to hold its relevance in generating cache for future uses.

If the Op de Kaart Application would extend to the rest of the Netherlands, a different cache with marker clustering can be used to make the visualisation comprehensive when zoomed out by preventing clutter. This cache contains the number of data points for each city per year. To add the possibility of scaling the visualization of Op de Kaart to larger areas, the caching algorithm should output two caching results: the number of data points per location per year, and the number of data points per city per year.

This thesis addresses the problem of the high latency of querying triple stores by investigating the case study of Op de Kaart, using data from the Utrecht Archives. The often-used solution of caching application objects to lower latency of applications based on triple stores is applied by creating and optimising an algorithm that can quickly cache the number of data points for each location per year retrieved from the Utrecht Archives, and the number of these data points per city. The maximum execution time indicated by the Utrecht Archives for this is one hour. The scalability of this algorithm is then evaluated by scaling the dataset size retrieved from Wikidata to simulate the addition of more locations to the Op de Kaart application.

This thesis will answer the question of how to create a caching algorithm for a triple store application that can be executed in a short time while handling a large number of data points and that can be scalable to a larger number of data points for future uses. The hypothesis is that to create such a caching algorithm, optimisation on the data retrieval method from the SPARQL endpoint will be necessary as the latency of the endpoint will slow down the algorithm the most.

A lot of literature can be found on improving the performance of SPARQL queries and triple store applications through query caching (Lorey and Naumann, 2013; Martin, Unbehauen, and Auer, 2010; Williams and Weaver, 2011; Zhang et al., 2016), but optimising the algorithm creating the cache itself is a less researched topic. By providing an answer to the research question, this thesis will contribute to the discussion on how to lower the common problem of high latency in web applications based on triple stores. Moreover, this project will contribute to the possibility of disseminating historical data and making it searchable in an intuitive way through a geographical visualisation.

2 Caching system and data description

2.1 Caching system

To create a fitting caching algorithm, the right caching system should be determined first (Gadkari, 2013). As the cache for Op de Kaart concerns frequently accessed data that is non-volatile, it has a temporal locality. Cache replacement algorithms like least frequently used resource (LFU), least recently used resource (LRU), and most recently used resource (MRU) are not suitable for this caching algorithm as all resources will be needed for low latency in the use of the application. Lastly, it will be a primed-cache pattern as the cache can be predicted in advance. In this case, the cache server can be primed and from there the application server can be populated in advance. Updating the primed cache can be done

in a routine schedule or an event-based mechanism (when new data is added to the Utrecht Archives).

2.2 Data description

For the right caching system the frequently accessed data needs to be determined as this will be the relevant data to cache in the application. In the case of Op de Kaart, this is the number of data points per location, per year. Upon scaling to a larger area visualisation, it is the number of data points per city, per year. In both cases, the data is retrieved from the first source: the Utrecht Archives. To be able to implement the visualisation for Op de Kaart, the data is enriched by data from the second source: Wikidata. From Wikidata the location name, coordinates, BAG id (basic register system of addresses and buildings), subject class and corresponding administrative territorial entity and their coordinates for all the locations from the Utrecht Archives are retrieved. The latter is only used in the post-processing of the data to get an accurate data point count per city.

Setting the right year format and removing duplicates could be done in the query itself. Duplicates were removed by filtering on Wikidata items, as each data point is linked to both Wikidata and BAG. Upon data inspection, however, 2,148 items were found in the dataset from the Utrecht Archives in which two years were given instead of one. For most cases, this implied a start and end date and therefore a period. For these items, the intermediary years have been taken along in the count for the corresponding location of the item in post-processing of the retrieved data. There were some cases in which the notation of two years was a wrong conversion of a date (01-01-1926 became 1126-1926), or the label included multiple year notations, leading to the wrong year notated (the publication year instead of the period to which the archival item belonged). Solving these errors in the original data is outside the scope of this thesis, but necessary for providing accurate search results per year for Op de Kaart.

Retrieving the right data point count per city was done in post-processing as well, as the data retrieved from the Utrecht Archives contained both streets and cities. For the street locations, the attribute 'located in the administrative territorial entity' could be used as this connects to the corresponding city. For the city locations, the same attribute connects to the province the city is located in. In those cases, the location name itself needed to be included in the count instead of the administrative territorial entity. For this, the subject class retrieved from Wikidata was used to determine whether the administrative territorial entity or the location name should be used in the count per city.

3 Method

The complete Python script of the caching algorithm, the SPARQL queries and the scripts used for the execution time measurements are available from my GitHub repository.

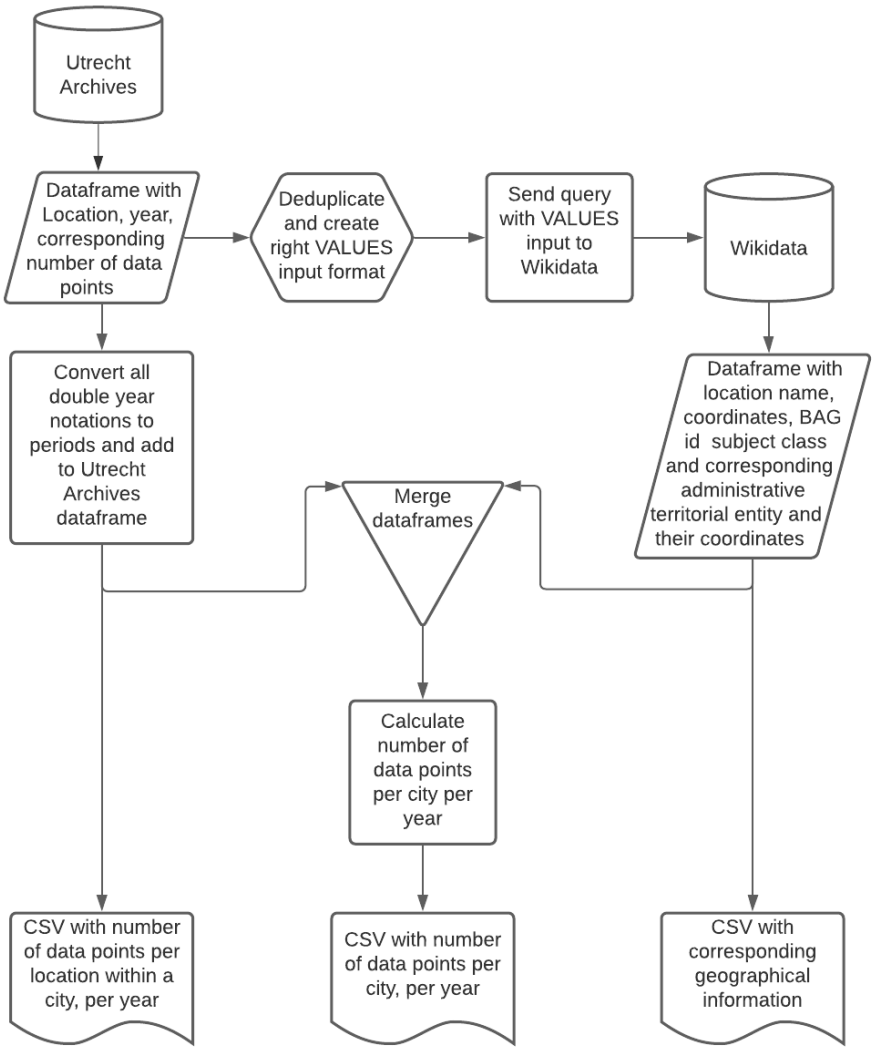

Figure 2: Flow chart visualizing the processes of the caching algorithm

3.1 Architecture of the caching algorithm

The processes of the caching algorithm are visualized in the Figure 2 above. The caching algorithm is a pipeline in Python, sending several POST requests to the "Dataset" SPARQL API from the Utrecht Archives and the SPARQL API from Wikidata. It is created as a pipeline for two reasons. Firstly, the limit of results retrieved from the Utrecht Archives SPARQL endpoint is 10,000 triples. Therefore, a pipeline is used to send requests in increasing offsets to get the complete set of results. Secondly, the attempted SPARQL query that retrieves both the count data from the Utrecht Archives and the corresponding attributes from Wikidata was prone to server time outs. This problem is bypassed by creating the pipeline in Python with

separate POST requests to both SPARQL endpoints and merging the data after retrieval.

The first POST request retrieves the number of data points per location per year combination along with the corresponding Wikidata id and year. A deduplicated list of the retrieved Wikidata IDs is used as the input for the VALUES operator in the SPARQL query that retrieves the location name, coordinates, BAG id, subject class, corresponding administrative territorial entity and their coordinates for each Wikidata id from Wikidata. All attributes but the location name and subject class are requested as optional to retrieve all locations. Both SPARQL queries can be found in Appendix A. The data is exported to three separate CSVs. Two CSVs contain respectively the number of data points per location within a city per year and the number of data points per city per year. As each location is recorded multiple times for each year in the first two CSVs, the corresponding geographical data of the locations is exported separately in the third CSV to prevent multiple recordings of the same data and decrease file size.

3.2 Performance test on scaling dataset

The performance measure on a scaling dataset is done on the scenario of an increase in data linked to new locations. This will give an insight into the extent of the scalability of the caching algorithm upon including increasing numbers of locations in the Netherlands in the visualisation of Op de Kaart. In this case, the ultimate goal would be the visualisation of the entire Netherlands and its locations with corresponding data points from the archives.

The performance test is done in two ways. Firstly, the scalability of the caching algorithm will be measured by increasing the Wikidata dataset in steps op 10.000 until the total number of streets recorded on Wikidata, 230,363, is reached. For this performance test, only the Wikidata dataset size will increase as the POST request to this endpoint proved to be the bottleneck in the execution time of the caching algorithm and will therefore be the best indication of scalability. Moreover, the Utrecht Archives do not contain enough data to simulate a scaling dataset sufficiently. The performance test is done on the same POST request to Wikidata used in the caching algorithm, but the values input is generated by Wikidata ids retrieved from Wikidata instead of the Utrecht Archives due to the earlier mentioned smaller dataset. The scalability of the algorithm is examined by measuring the execution time of the algorithm for each increase in the number of data points retrieved and visualizing this in a performance graph. The algorithm is considered to scale well when it maintains the same efficiency when the workload increases, which would mean that the execution time has a linear relationship to the number of data points queried (Bertossi, Pinotti, and Gupta, 2010). To test this, the results of the performance tests are tested for linearity by attempting to fit a linear regression model to them.

Additionally, an insight into the extent of the scalability of the algorithm is given by measuring the execution time of the full algorithm when retrieving all 230363 streets from Wikidata. To simulate this scenario as accurately as possible, 50 data points will be retrieved from the Utrecht Archives for each location retrieved from Wikidata.[1] This way, the retrieval, processing and export of those extra data points are taken along in the measured execution time as

[1] The decision for 50 data points for each location is based on the average number of data points per location currently retrieved from the Utrecht Archives.

well.

3.3 Measurement of the results

All execution times referred to in the following result section are measured with the *timeit* module in Python. All benchmarking is done on a machine with the following configuration: Python 3.8.5 in Ubuntu 20.04, on an Intel Core i7-7820X with 64GB of RAM and a 500 GB SSD, over a wired internet connection, measuring 85 Mbps download speeds and 95 Mbps upload speeds, with a ping of 2 ms. To get an accurate time measure, all separate parts of the code were run 50 times to get a mean execution time, standard deviation and maximum execution time. For the performance test on scalability and the full algorithm scaled to all streets in the Netherlands and corresponding data points, the code was run respectively 20 or 10 times due to practical time constraints. The individual execution times for the POST requests to the Utrecht Archives and Wikidata are manually inspected to check if the results are not influenced by the query result being cached by the SPARQL endpoints after the first POST request, lowering the following execution times and iteratively skewing the results.

4 Results

4.1 Total execution time

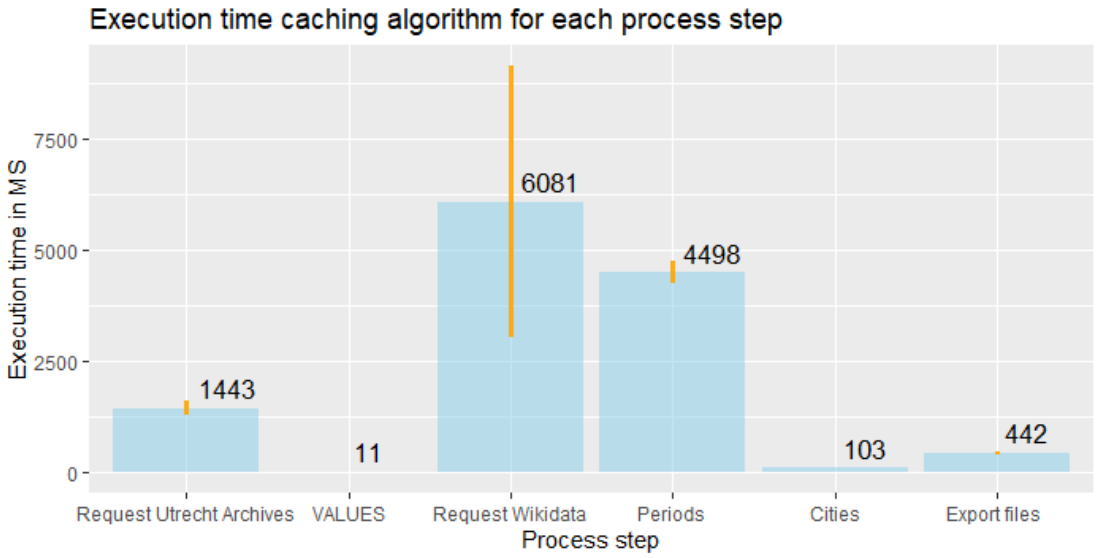

Figure 3: Barplot with the mean execution times and standard deviation of the caching algorithm

The measured mean execution times, standard deviation and maximum execution times for all separate processes of the caching algorithm and its calculated total results can be seen in Figure 4 above. The total execution time of the caching algorithm is averaged 12,578 milliseconds with a standard deviation of 3,169 milliseconds, querying 89599 data points from the Utrecht Archives and 5268 data points from Wikidata. On manual inspection no indication could be found that the query results were cached by the the Utrecht Archives and Wikidata endpoints, as the first measured execution time was not longer than the following measurements.

The biggest bottleneck in the execution time of the algorithm is the POST request to Wikidata with 6,081 milliseconds, which is much longer than the 1,443 milliseconds for the POST request to the Utrecht Archives. The reason for this is that the request to Wikidata retrieves many more attributes for each VALUES input. The POST request to Wikidata also has a relatively high standard deviation of 3,058 milliseconds.

Another bottleneck in the algorithm is the conversion of the double year notation to years, with 4,498 milliseconds. This is caused by the use of 'for loops' to iterate through the data frame row for row to converse the double year notation to periods.

4.2 Scalability

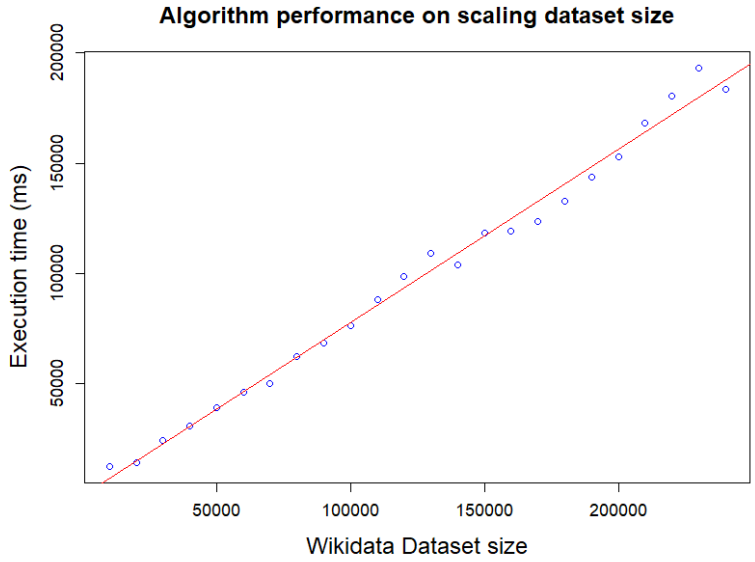

Figure 4: Performance graph on scaling dataset size

The caching algorithm can scale up to 60,000 results retrieved from Wikidata within an averaged 46,091 milliseconds. After this, the VALUES operator reaches an input limit and the code needs to be changed. By dividing the input for VALUES into smaller parts and sending them separately, the limit can be traversed. In total, retrieving all the streets in the Netherlands recorded on Wikidata takes an averaged 183,391 milliseconds, with a standard

deviation of 14,756 milliseconds. The measured mean execution time, the standard deviation and maximum execution time for each dataset increase can be found in Table 2 further below.

The performance of the algorithm upon an increasing dataset size retrieved from Wikidata is visualized in Figure 2 above. The blue dots indicate the measured time for each dataset size increase, the red line is the regression line best fitted to the plot. The linear regression model has a highly significant p-value of 2e-16 and an adjusted R-squared of 0.9902.

The execution time of the full algorithm with all 230,363 streets from Wikidata is averaged 903,122 milliseconds with a standard deviation of 15,216 and a maximum execution time of 931,151 milliseconds. This comes down to an average execution time of 15 minutes.

Dataset size	Mean (ms)	SD (ms)	Max (ms)
10,000	12,482	11,376	46,700
20,000	14,038	2,328	18,597
30,000	24,226	9,592	62,405
40,000	30,749	3,440	39,085
50,000	39,069	8,643	68,430
60,000	46,091	6,625	59,980
70,000	50,289	8,596	79,195
80,000	62,255	11,852	105,255
90,000	68,465	11,523	102,643
100,000	76,302	8,774	96,169
110,000	88,045	14,208	117,838
120,000	98,524	11,578	124,749
130,000	109,086	10,353	126,347
140,000	103,753	11,208	127,276
150,000	118,502	21,433	181,275
160,000	119,307	15,989	161,356
170,000	123,443	13,315	164,102
180,000	132,720	19,739	190,607
190,000	143,920	15,110	185,472
200,000	152,865	13.117	192,079
210,000	168,203	18,358	212,681
220,000	180,478	18,665	221,325
230,000	193,236	20,085	233,009
240,000	183,391	14,756	208,283

Table 1: Table with measured execution times for each Wikidata dataset size increase

5 Conclusion and discussion

This thesis investigated the question of how to create a caching algorithm for a triple store application that can be executed in a short time while handling a large number of data points. The hypothesis was that specific optimisation on the data retrieval method from the SPARQL

endpoint would be necessary to make the query fast enough to be implemented regularly. In creating the caching algorithm, the biggest challenges were indeed found in retrieving the data from the SPARQL endpoint of the Utrecht Archives and Wikidata. Yet these were not constraints in speed, but practical ones. As a solution to the limit of 10,000 on the results from the Utrecht Archives SPARQL endpoint, and the server time outs when querying on the Wikidata SPARQL endpoint, a Python pipeline was created that could send all requests iteratively. With this pipeline, all needed results can be retrieved in one process and the code is easily scalable to larger datasets. The high latency of querying on a SPARQL endpoint is still influenced by the execution time results of the caching algorithm, in which the POST request to the Wikidata SPARQL API was the bottleneck. Moreover, its relatively high standard deviation is a sign that its execution time fluctuates considerably from run to run, leading to a lower consistency in the total execution time of the caching algorithm. However, as a caching algorithm for the current data visualisation from Op de Kaart with an execution time of 12,578 milliseconds, no further optimisation is needed to be able to implement the algorithm regularly to prevent an outdated cache. Furthermore, the performance test on the scaling Wikidata dataset indicated a linear trend in the performance data, which means that the algorithm scales well as it maintains the same efficiency when the workload increases. Moreover, if the Op de Kaart application will extend to a larger area, the other cache with the count per city can be used for the visualisation. In the scenario of including all documented streets in the Netherlands with 50 archival items per street, the caching algorithm would need 15 minutes to retrieve, convert and export all data.

Of course, much can still be done to further optimise the implementation of cache in a triple store application like Op de Kaart. For instance, all data retrieval from the SPARQL endpoints is done iteratively while the execution time can be further improved by sending parallel POST requests instead. The execution time of the caching algorithm for the current data visualisation from Op de kaart does not necessarily need any further optimisation, but when retrieving much larger datasets, parallel requests could be of real value. Furthermore, as mentioned earlier in Section 2, the data from the Utrecht Archives contained cases in which the year notation is wrongly conversed, leading to inaccurate dates connected to the data points retrieved by the caching algorithm. Consequentially, a visualisation based on the current cache retrieved from the Utrecht Archives would be somewhat inaccurate. More importantly, the application would not be able to provide fully accurate search results per year. Resolving the wrong data conversions in the data from the Utrecht Archives and likely many other archives would be an interesting challenge. To get an accurate view of the scale of this issue, stocktaking should be done by manually inspecting archival items with a beginning and end date far from each other as this often indicated a wrong conversion. Ideally, an automated method would be used to get more accurate time conversions from the descriptions of the archival items. Still, if there are only a few cases manual fixes would be more feasible. Another interesting implementation of caching would be to further follow the approach of Martin, Unbehauen, and Auer (2010) and cache query results as well. An example would be caching the first five images of locations in the inner city of Utrecht, like the Domplein or Janskerkhof. Furthermore, Lorey and Naumann (2013) presents different methods to retrieve data that is potentially interesting for following requests in advance. An example of this would be retrieving the first five images of the locations surrounding the

selected location. Frequently accessed data could be identified and cached to lower latency in data retrieval of these locations, and corresponding data points from surrounding locations can be retrieved in advance to further improve the user accessibility of the application.

References

Abbott, H. P. (2020). *The cambridge introduction to narrative*. Cambridge University Press.

Berners-Lee, T. (2006). Linked data-design issues. *http://www. w3. org/DesignIssues/Linked-Data. html*.

Bertossi, A. A., Pinotti, M. C., & Gupta, P. (2010). Scalable algorithms for server allocation in infostations. *Handbook of research on scalable computing technologies* (pp. 645–656). IGI Global.

Bondi, A. B. (2000). Characteristics of scalability and their impact on performance. *Proceedings of the 2nd international workshop on Software and performance*, 195–203.

Gadkari, A. (2013). Caching in the distributed environment. *Advances in Computer Science: an International Journal*, *2*(1), 9–16.

Horak, L. (2016). Using digital maps to investigate cinema history. *The Arclight Guidebook to Media History and the Digital Humanities*, 65–102.

Lorey, J., & Naumann, F. (2013). Caching and prefetching strategies for sparql queries. *Extended Semantic Web Conference*, 46–65.

Martin, M., Unbehauen, J., & Auer, S. (2010). Improving the performance of semantic web applications with sparql query caching. *Extended Semantic Web Conference*, 304–318.

Rusher, J. (n.d.). Rhetorical device: Triple store. https://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/rusher.html

Teng, S.-H. (2016). Scalable algorithms for data and network analysis. *Foundations and Trends® in Theoretical Computer Science*, *12*(1–2), 1–274.

Verhoeven, D., Bowles, K., & Arrowsmith, C. (2009). Mapping the movies. reflections on the use of geospatial technologies for historical cinema audience research.

Williams, G. T., & Weaver, J. (2011). Enabling fine-grained http caching of sparql query results. *International Semantic Web Conference*, 762–777.

Zhang, W. E., Sheng, Q. Z., Qin, Y., Yao, L., Shemshadi, A., & Taylor, K. (2016). Secf: Improving sparql querying performance with proactive fetching and caching. *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 362–367.

Appendix A. SPARQL queries

Retrieving the number of data points per location per year from the Utrecht Archives

```
SELECT  *
{
  {
    select ?street ?year (count (?year) as ?count) where
    {
        ?sub sw:hasBeginTimeStamp ?date.
        ?sub dct:spatial ?street.
                BIND (YEAR(?date) AS ?year)
                FILTER regex(?street, "wiki")
        }
        group by ?street ?year
    order by ?year
  }
}
```

Retrieving corresponding geographical information from Wikidata

```
SELECT ?street ?streetLabel ?coords ?bagID ?city ?cityLabel
?citycoords ?s_class WHERE {
  VALUES ?street {}
  OPTIONAL {?street wdt:P625 ?coords.}
  OPTIONAL {?street wdt:P5207 ?bagID . }
  OPTIONAL {?street wdt:P131 ?city . }
  ?street wdt:P31 ?s_class .
  ?city wdt:P625 ?citycoords.
  FILTER(?s_class = wd:Q79007 || ?s_class =  wd:Q2039348 ||
         ?s_class = wd:Q3957)
  SERVICE wikibase:label
          { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],nl". }
}
```