Interactive geographic visualization of very large GLAM data

Brekelmans, J.A.

Award date:
2015

Department of Mathematics and Computer Science
Section Algorithms and Visualization

Interactive Geographic Visualization of very large GLAM data

Master Thesis

Jasper Brekelmans

Supervisors:
Prof. Dr. B. Speckmann
Dr. K. Verbeek

Eindhoven, May 2015

Abstract

Libraries of today provide public access to millions of bibliographic records with metadata such as publish locations and holdings. Users of these datasets such as librarians and researchers need to analyse and maintain these datasets, but most libraries only provide textual interfaces for these purposes. Large tables and other text lists require paging, are unsuited for high level overviews and are difficult to navigate. This problem can be overcome by specialized tools that realize insightful, intuitive and comprehensive visualizations through the use of graphical overviews. GlamMap is a geo-spatial visualization tool that allows users to visualize metadata of cultural heritage artifacts, such as place of publication and year of unveiling, on an interactive two-dimensional geographical map.

This thesis discusses the integration of Trove, a dataset with more than 60 million books aggregated from libraries throughout Australia, with GlamMap. We use a relational database to optimize storage for retrieval of data. The key part of our visualization is showing hierarchical clusterings of datasets on a geographical map. Supporting various kinds of filters for high level overviews at interactive speeds proves the most difficult to achieve.

Preface

I would like to thank my supervisors prof. dr. B. Speckmann and dr. K. Verbeek for their enthusiasm, commitment and ideas and letting me share this experience with them. I would like to additionally thank other creative and intelligent minds involved directly in the development of GlamMap or one of its brainstorm sessions for letting me share the experience. Furthermore I would like to thank my family for giving me this opportunity and my friends, family and colleagues for their support.

Contents

Chapter 1

Introduction

Digital libraries of today contain millions of records. Without specialized tools, librarians and researchers cannot easily maintain and understand these massive datasets [8, pp.1–9]. Most libraries only provide textual interfaces for these datasets. Although they do allow users to perform a variety of queries, they do not provide good general overviews of search results. Typically results consist of large tables or other textual lists that do not fit on the screen and therefore require paging. These interfaces are difficult to navigate, provide no general overview of libraries' holdings and finding a particular record can be time-consuming [7, 27]. More graphical representations are known to provide quick access to bibliographic data, reduce search times, facilitate browsing and identification of relevant metadata and provide a quick overview of the coverage of libraries [8].

Visualizations exist to solve these problems through the use of intuitive and clear overviews that are compact yet complete. Graphical interfaces allow users to browse datasets quickly, reduce search times and find relevant information easily. In the case of bibliographic metadata the coverage of libraries can be established quickly through visualizations [7, 8], but no visualizations providing general overviews exist yet. By displaying bibliographic records geographically on maps, overviews are intuitive. For example: an overview of the coverage of libraries is naturally encoded in the geospatial distribution of bibliographic records. Using geographic metadata to visualize large datasets guides humans by enabling querying on a geographical scope [8, pp. 171–179].

GlamMap is a tool that realizes a two-dimensional interactive geospatial visualization of bibliographic datasets and allows users to find (relevant) records quickly through a map-based overview. Metadata such as place of publication, year of publication, title and author(s) of bibliographic records are encoded in the visualization, but any dataset with similar fields is supported, in particular Gallery Library Archive and Museum (GLAM) metadata. The main overview of GlamMap shows clusterings of its data on their geographical location. Due to its flexibility in datasets, GlamMap is envisioned as a publicly and easily accessible website that allows users to upload their own datasets, but only a prototypical version of GlamMap supporting one dataset exists.

While showing small datasets such as Risse's *Bibliographica Logica* (consisting of roughly 7000 records) in GlamMap is perfectly possible, visualizing datasets with millions of records in

GlamMap has not been attempted until now. Examples of such very large datasets are: Trove [4], a publicly available dataset maintained by the National Library of Australia (60 million books with a total of more than one-hundred-million copies), and WordCat [24], a dataset maintained by Online Computer Library Center (300 million records in 2013).

1.1 Problem analysis

The problem is to integrate Trove with GlamMap, which is two-fold: overcoming data-challenges and adapting our visualization to Trove.

Our map-based visualization requires that maps have the appropriate "fullness" [28, p. 315]. This means we have a good balance between showing information on top of the map and seeing enough of the map itself, such as that in Figure 1.1. GlamMap uses *glyphs*, each encoding information of the dataset, on top of maps. Clustering glyphs is the main tool used to achieve the appropriate fullness. Overlap of glyphs on the map is not desired since this inhibits users from easily reading the visualization. If we only show one glyph per item then a map also may not have the right fullness. For instance: when thousands of glyphs are inside the *viewport*, the visible geographic region of the map, they most likely have to be drawn very small so that they do not overlap and the map is not too full but then any information they encode cannot easily be seen. Regions of the map visible in the viewport may be of different sizes because users can zoom in or out and resize the viewport. When zooming in a glyph should shrink, otherwise its area may grow too large. Similarly, when zooming out a glyph should grow, otherwise it may become too small and too difficult to see. Ideally the size of a glyph is proportional to a constant-sized viewport and thereby independent of zooming. Note that when zooming in and out the amount of cluttering increases and decreases, respectively, and unless action is taken glyphs will overlap. A clustering therefore has to be adaptive to zooming by the user but must be so in a consistent fashion. The latter property is desired because it supports the preservation of the mental map of a user over zooming.

We require that GlamMap is transformed from a prototype into a website using a client-server architecture. With this setup GlamMap is available to any user with an internet connection and a device with a web-browser. More importantly we can utilize the dedicated hardware of servers when storing and processing the large amount of data in Trove, and allow for horizontal scaling (adding more servers if necessary). When making datasets available online, it is also much easier for users to collaborate in the exploration and investigation of datasets and any number crunching can be offloaded to the server. The user therefore does not rely on the capabilities of his or her own device, as long as it is capable of running a web browser. GlamMap will be much more responsive for less-capable devices this way. Consider on the contrary that the user is required to download the entire dataset he or she wishes to visualize. When visualizing Trove, the user would then be expected to download Gigabytes of data, which may take much time, even if one does not count the time spent computing and rendering glyphs. Slow devices will likely not be capable of handling this very large data. Number crunching such as computing clusterings is ideally computed server-side. This way the *client* (the computer of a user of this web page) does not need to load the entire dataset to show the initial overview on the map, but only the

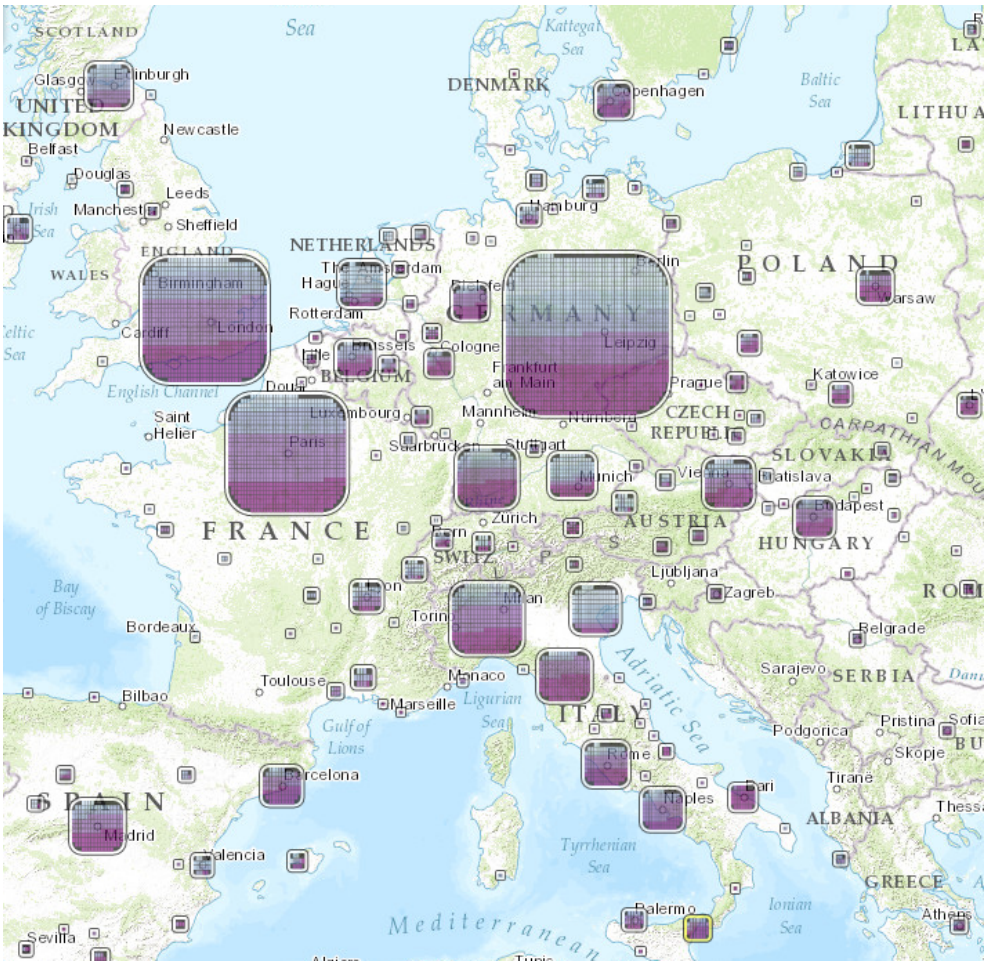


Figure 1.1: An overview of our map-based visualization.

clustering itself. This will make GlamMap more responsive when first loading the web-page because the clustering of Trove can be represented in 3 Megabytes whereas the dataset itself is several Gigabytes. Even if the hardware on the client is just as capable as the server, we can run native code on the latter (e.g. compiled C++) as opposed to generally slower JavaScript on the client. One might argue that the additional overhead of sending the clustering from the server to the client outweighs the faster computation of the clustering, but when initially showing the overview on the map this transfer replaces having to load the entire dataset on the client.

1.2 Related work

The task of providing insight into large spatio-temporal datasets with different attributes is challenging and therefore not applied often in many application domains [18]. In particular, libraries often only provide textual interfaces for collections of books with metadata such as the year and location publishing and visualization tools are needed [6]. As a result, many visualizations are aimed to make analyzing and studying metadata of records in digital libraries easier. Some of these visualization tools use a hierarchical structure in the metadata (e.g. the Dewey Decimal classification). Others simply plot this metadata on meaningful axes [23, 27]. Compact visualization of hierarchies can also be attained by treemaps, constructed by recursively splitting rectangles according to a hierarchical structuring of the data [12].

We follow the current trend in web technologies to build a system within a rapidly changing unpredictable landscape of information processing agents [32].

In some domains geographical metadata plays a significant role. GlamMap is targeted on such a domain and users need to be able to specify a geographical scope. Simple shapes on maps intuitively encode the distribution of items over geographical regions. There are very few tools, however, using map-based visualizations for bibliographic datasets. GeoVIBE [9] is a tool showing a collection of documents (books) on a geographical map, using location metadata. But the functionalities of GeoVIBE do not scale well to larger datasets such as Trove. Recently, library data providers like DPLA [14], OCLC (WorldCat [24]), and Europeana [16] have begun to develop map-based interfaces to explore and analyze their collections. Although early prototypes of these interfaces have already proven to be very useful to users, in terms of visualization techniques they are very limited. In fact, in reaction to a demonstration of an early prototype of GlamMap, OCLC has decided to start a new project in collaboration with GlamMap developers to develop a new map-based interface for WorldCat based on GlamMap.

The Visual Information-Seeking Mantra induces guidelines on user interfaces where users customize a high level overview by analyzing the overview and setting filters iteratively [26]. This type of visual analysis, that proceeds in a refining manner, can be a highly effective means of exploring datasets [22], [17] and assists librarians and researchers in their jobs [6].

Dynamic graphics on maps such as in GlamMap provide enormous potential for visualizing spatial data [15]. GlamMap's map visualization problem resembles dynamic map labeling [5]. However instead of hiding labels of documents we compact them by merging. Similar techniques have been used before (see e.g. [25], which includes a good overview). Merged labels (glyphs), or rather an aggregation, changes as users zoom in or out. To aid the preservation of

the user's mental map in GlamMap, hovering the mouse pointer over a glyph previews the (sub)-aggregations of that glyph on the next zoom level. This technique resembles that of semantic lenses [20] or related techniques [19].

Chapter 2

Initial state of GlamMap

GlamMap is a prototypical visualization tool used for visualizing bibliographic datasets that follows popular user interface guidelines discussed in the Visual Information-Seeking Mantra [26]. GlamMap is a standalone web page. An overview of the Risse dataset in GlamMap can be seen in Figure 2.1. The middle region shows a clustering on a geographical map and the panels around it allow for filtering and inspecting data. Sections 2.1 through 2.5 discuss the visualization and user interface of GlamMap in more detail. At this point some efforts to turn GlamMap into a client-server application were made, as discussed in Section 2.6. Recall visual elements in GlamMap are called glyphs and each glyph represents bibliographic records close to each other and provides a high level visual representation for these records.

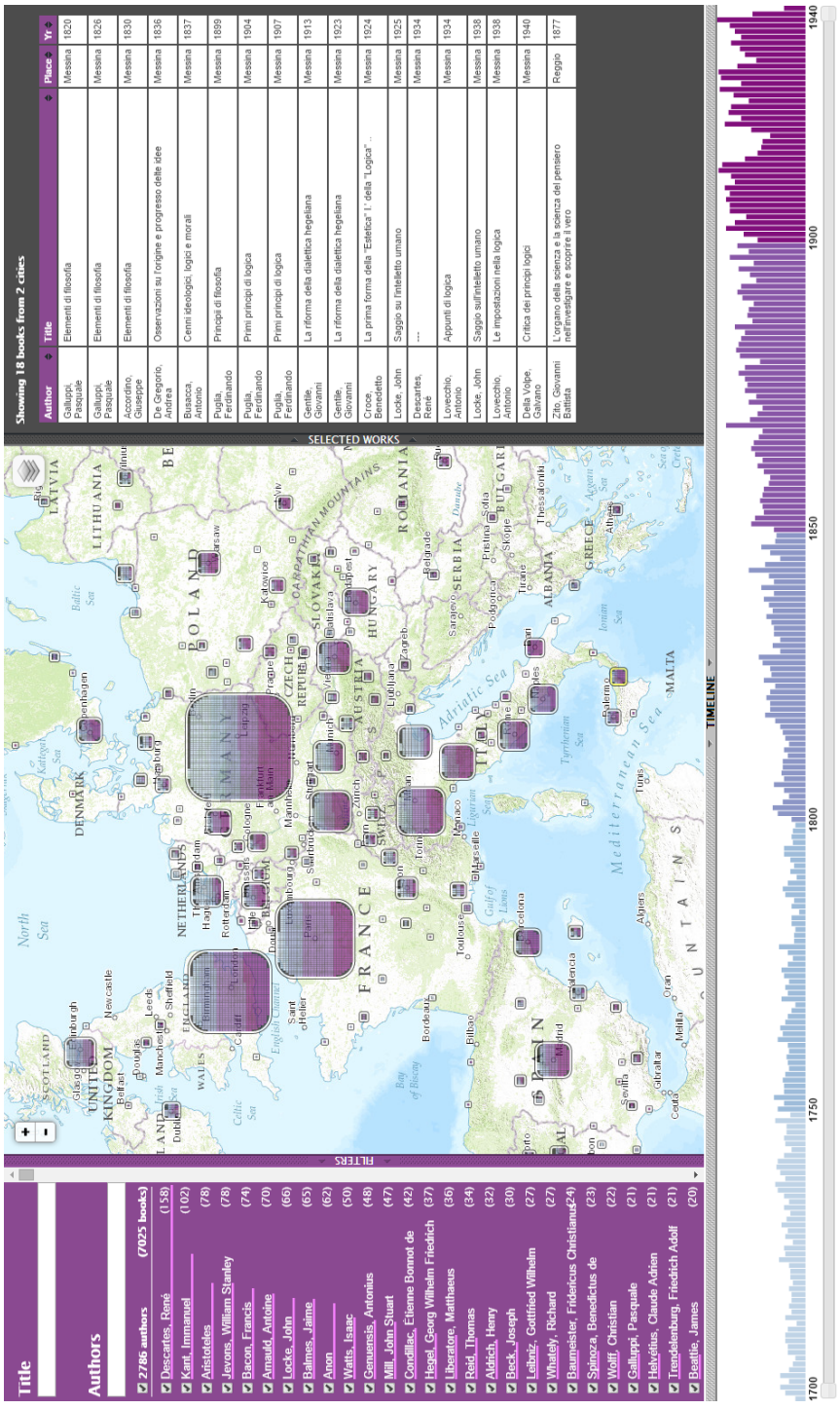


Figure 2.1: An overview of the old user interface of GlamMap when showing Risse's dataset.

2.1 Maps

GlamMap is a web-based visualization tool and supports numerous different geographical maps, on which any visual elements representing the dataset are overlain. Maps are important because they provide information about geographical regions to the user and allows them to orient themselves. The most suitable map may change from dataset to dataset. For example: in Figure 2.2 a historic collection of music scores from the years 1781-1848 is shown on a map from the same time period. Using a map reflecting the administrative subdivision of today would not show the distribution of items over the different empires at that time.

A digital map is a set of informational drawings of a rectangular geographic region in different levels of detail. Depending on the zoom level at which the user wants to see a region, a drawing with the appropriate level of detail is chosen. The zoom levels at which a digital map can be viewed are usually discrete. Typically there is one drawing for each zoom level when not considering extremely high or low zoom levels. As drawings of the entire map in high levels of detail can become extremely large, it is desired to only have the portion of the drawing the user can see at hand so that less storage is required. This principle applies directly to websites since it is required that maps are downloaded. We only have to download the portion of the map that users can see. For this reason drawings of a map are subdivided into *tiles* and only the tiles that are visible are loaded. The more technical term for referring to maps is therefore *tilesets*.

GlamMap can support arbitrary tilesets and currently allows the user to choose among some tilesets provided and hosted by ArcGIS [1], a mapping platform. These tiles contain the oceans, place names and geographical regions. In fact they contain everything shown on maps in GlamMap but glyphs. The software library used for showing tilesets is Leaflet [3]. Zooming is done in discrete steps, like most other map libraries, and when moving from zoom level i to $i+1$ the width and height of the area represented by the viewport are halved. The area represented by one tile therefore decreases fourfold, but the same number of tiles can be seen in the viewport. Two different tilesets are compared in Figure 2.3. For regions familiar to the user the top tileset provides a cleaner and less tiresome to interpret overview than the bottom tileset. The bottom tileset shows city names and elevation data and better suits exploration of the dataset around regions less familiar to the user.

Map projection. Map projection is the process of projecting geographical coordinates to points on a flat rectangular surface. Web-based digital mapping libraries often use Web-Mercator projection. Here latitude-longitude pairs on an ellipsoidal model of Earth are treated as if they are coordinates on a sphere and scaled to a square where $(0,0)$ is the upper-left corner and $(256,256)$ is the lower-right corner. This is the de facto standard for Web mapping applications and is used by Leaflet, and widely by Google Maps, Bing Maps and OpenStreetMap.

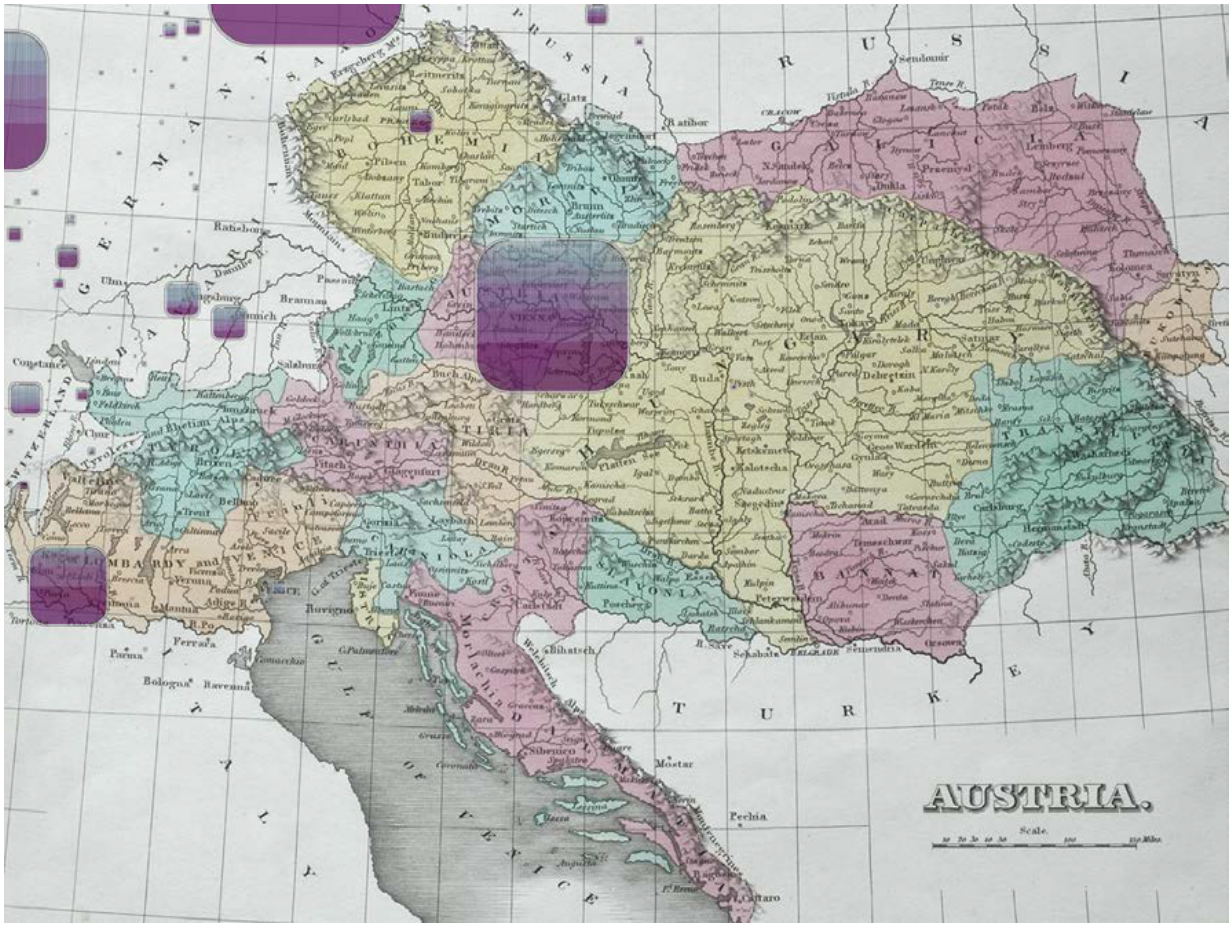


Figure 2.2: Musical scores from the years 1781-1848 according to the dataset of WorldCat [24], visualised on a map of the Austro-Hungarian Empire (original source [6, Figure 7]).

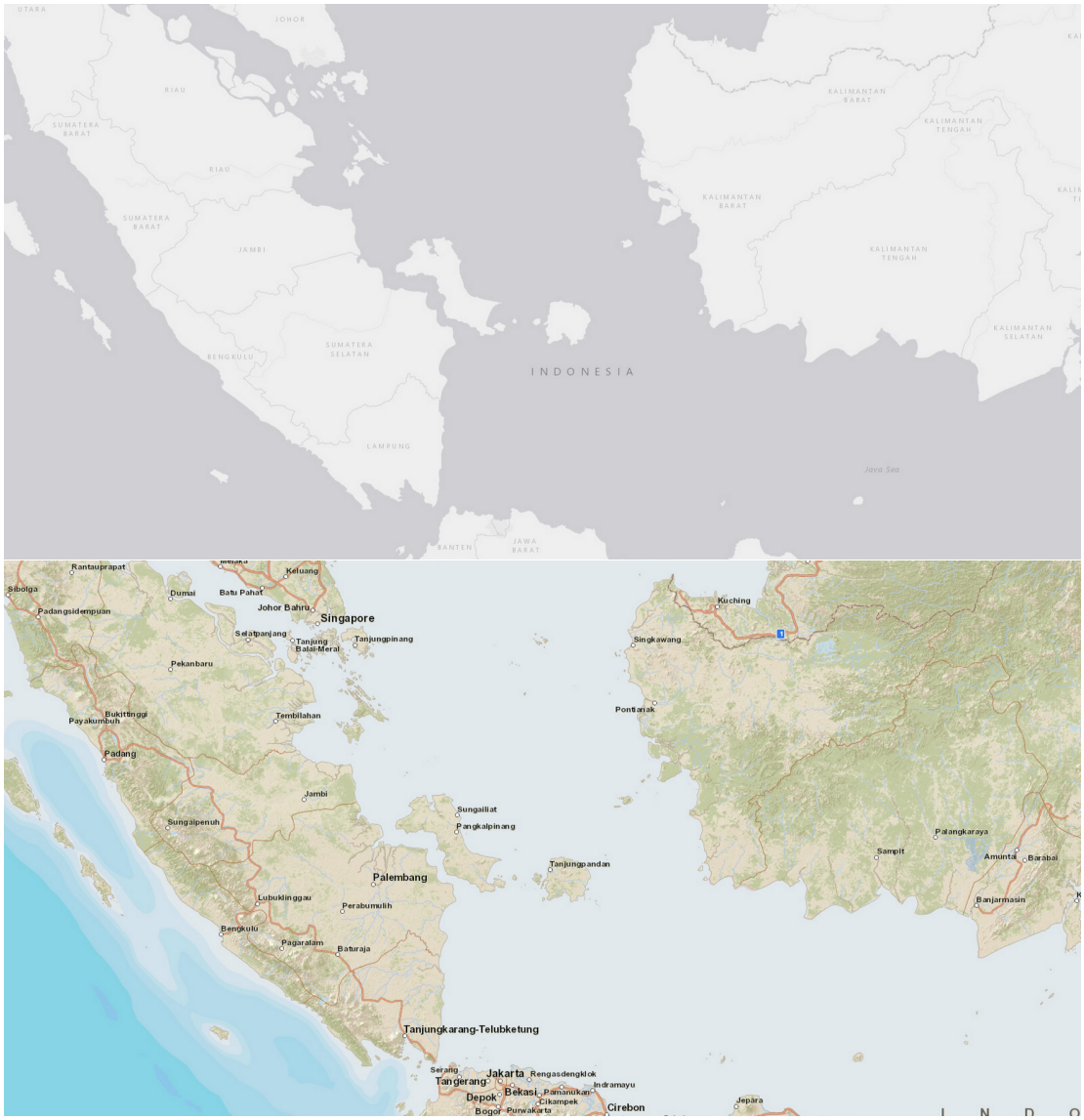


Figure 2.3: Two tilesets showing geographical information on different levels of detail. The bottom tileset includes elevation data, main roads and place names whereas the top tileset does not.

2.2 Search interface

GlamMap [2] realizes the Visual Information-Seeking Mantra [26]: overview first, zoom and filter and detail on demand. The Mantra is a widely accepted framework for visual interface design and applying it will likely result in an efficient-to-use interface. An overview of the entire dataset can be seen initially, which shows where books have been published. For each location we show a glyph and within this glyph we categorize bibliographic records by year. The map supports interactive zooming, panning and detail on demand. In Figure 2.5 the *year* panel shows the distribution of items in the Risse dataset over years of creation (a histogram). It allows a range filter over these years to be specified using a scented widget [33], which provides visual cues (how many books are being selected) to the user while selecting a year range. In Figure 2.4 the *creator* panel is shown for the Risse dataset. It allows users to filter on creator (e.g. authors of books) and title. By clicking the topmost checkbox all authors can be selected and unselected (for inclusion in the visualization) simultaneously. The year filter affects the creator panel, but not vice versa. The year and creator panel both affect the clustering shown on the map as well as the details-on-demand panel. The information that becomes visible when a glyph is selected is shown in the details-on-demand panel, and can be seen in Figure 2.6 (Risse dataset). For each book represented by the selected glyph, the details-on-demand panel shows the author, title and exact year of publication. When no glyph is selected the details-on-demand panel allows users to page through all books.

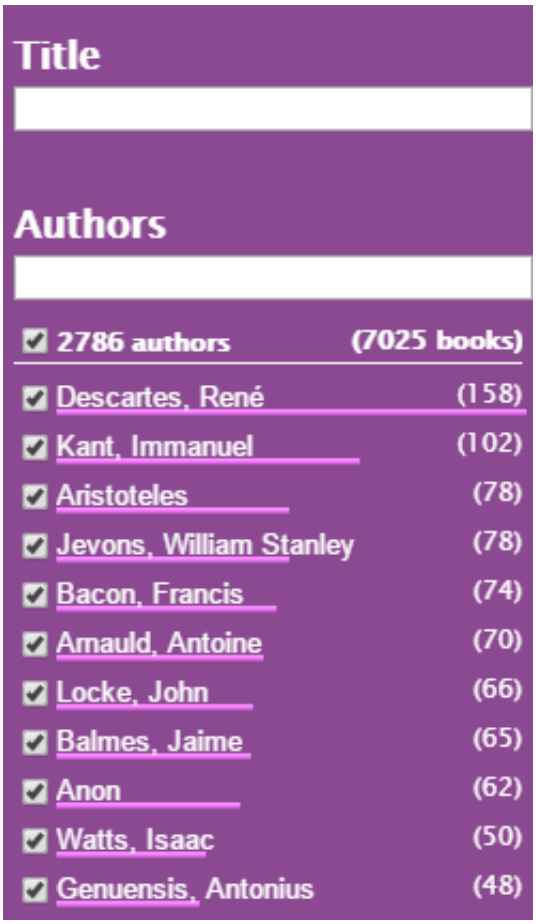


Figure 2.4: The creator panel shows a list of creators (e.g. authors) in order of the number of items they have produced.

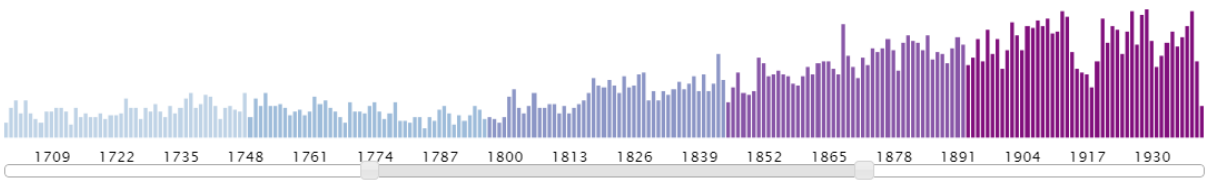


Figure 2.5: A timeline showing the amount of items per year (e.g. when a book was published) and allowing the user to specify a minimum and maximum range filter.

Author	Title	Place	Year
Baumgarten, Alexander Gottlieb	Acroasis logica, aucta et in systema redacta a Jo. G. Töllnero	Halle	1773
Beattie, James	Essay on the nature and immutability of truth	London	1773
Beattie, James	Verhandeling over de natuur en onveranderlykheid der waerheid	Utrecht	1773

Figure 2.6: The details that become visible when selecting a cluster. All items represented by the selected cluster are shown in a paged table. The author, title, place and year are shown for all items.

2.3 Glyphs

Glyphs are visual elements overlaying the geographical map and they are the encoded form of bibliographic items (records). Since we use maps, glyphs naturally should encode the geospatial distribution of these items. An instance of a glyph is shown in Figure 2.7. Each glyph is shaped like a rounded rectangle, has a thin border and the items it represents each correspond to a tall rectangle. These rectangles are laid out in an evenly spaced grid on a black background. The *interior* of a glyph is the region within its border. A glyph is drawn slightly transparent so that the underlying map can be seen better and not too much emphasis is placed on the glyphs.

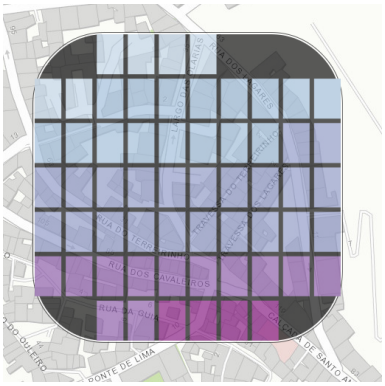


Figure 2.7: A glyph representing 60 bibliographic items.

Glyph shapes. To easily be able to read the density of items from the visualization, the shape of a glyph must be simple. Proportional symbol maps usually use a circle because they are compact and visually stable (cause little eye wandering) [13, p. 175] and are preferred by users [29]. Squares are perceived negatively by users as “ugly” and “angular” [29]. Users can however estimate the area of squares better than the area of circles. The shape used for glyphs is a mix of the two, specifically a rounded rectangle.

Glyph border. The purpose of a glyph border is to produce a high-contrast along the edge of a glyph and to give glyphs a minimum size. These two are necessary to make a glyph more easily

distinguishable from the underlying map. Due to a sufficiently thick border small clusters are easily pointed at with the mouse pointer, making interaction easier.

Figure 2.8: A glyph representing 5836 bibliographic items.

Glyph interior. The glyph interior shows one tall rectangle for each item the glyph represents and its color is used to encode item attributes. There are many different ways to choose this color encoding (e.g. the year in which books are published, and the author of a book), but in any case items are partitioned into categories and each category corresponds to one color. The individual rectangles can clearly be seen on glyphs with a small amount of items, as in Figure 2.7, but are harder to see on large glyphs such as the one in Figure 2.8. Items in the interior are sorted by category so it is easy to see the distribution of items over categories. For example: it is easy to see how large one category is relative to another. If one does not apply sorting the glyph interior will look like noise. In this figure as well as Figure 2.7 a color encoding based on a sequential categorization of books into their year of publishing is shown.

2.4 Hierarchical clustering

A clustering is a tree of glyphs that encodes at which zoom level glyphs split into smaller glyphs when zooming in. More formally a hierarchical clustering is a tree where each node n represents a cluster $c(n)$ and the zoom level $z(n)$ at which $c(n)$ should be split into the clusters of children of n. For each zoom level z we can perform a breadth-first search where descendants of any node n are not traversed if $z(n) \leq z$. The nodes traversed this way, except those nodes of which children have been traversed, are the nodes of which clusters are to be shown at zoom level z. GlamMap has an $O(n^2)$ algorithm that constructs this tree in a bottom-up fashion. The algorithm works as follows: given that clusters shrink on the map when we zoom in, there is a zoom level at which none of the initial (leaf) clusters intersect (assuming they all have distinct locations). Now we zoom out and handle any intersections that occur by merging intersecting clusters.

A cluster in this visualization is the algorithmic representation of a glyph. While in general the information determining the size of a glyph depends on the application, for GlamMap it is determined by the number of bibliographic items represented by a cluster. Recall a glyph shows a categorization (partitioning) of records into five categories, each with a different color, as shown in Figure 2.9. When merging clusters, this partitioning needs to be preserved and in fact we store five integers per cluster. Stated differently: the number of items is stored for each *bucket*. All items with the same category are placed in the same bucket.

A clustering should maintain the right map "fullness" when zooming in or out. If we do not shrink clusters when zooming in then the map may become too full. Similarly, if we do not grow clusters when zooming out then the map may become too empty and glyphs may no longer be large enough to see. For this reason the size of glyphs is independent of the zoom level. To prevent overlap and maintain the right "fullness" we merge and split clusters. When zooming in,

if splitting a cluster x will not cause overlap, then x will always split into its children. Similarly, when zooming out, if we do not merge clusters X with the same parent x and this causes overlap then X will always be merged into x.

Global. It is important to compute the hierarchical clustering globally, that is: the input items for this computation should not be restricted to only those items inside the viewport. Doing otherwise will cause different clusters to be computed for a fixed region of the map depending on how much of the surroundings of that region are visible in the viewport. In other words: we do not want panning to potentially completely change the clustering. Computing a clustering once (per set of items) also enables its hierarchy to be reused and we do not have to recompute the clustering when panning, improving the response-time of the visualization.

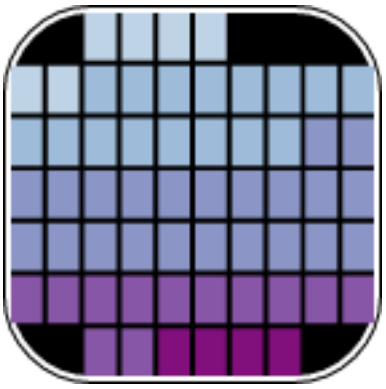


Figure 2.9: A cluster that has its records categorized into five buckets.

Map projection. The clustering algorithm like the user interface also applies map projection, but here we use Mercator instead of Web-Mercator. This is a deficiency of the old GlamMap implemented either to save time or incorrectly, but it cannot be easily seen for the Risse dataset and is not significant.

Agglomerative. Hierarchical clusterings shown in GlamMap are computed in a bottom-up fashion. Since the location of a cluster should reflect the locations of its items, we need to know child clusters to compute parent clusters. More precisely: when merging two clusters c_1 and c_2 into merged cluster c_3, the location of c_3 is the weighted average of the locations of c_1 and c_2 (each location is a latitude-longitude pair), where the weight of a cluster is the number of items is represents. We use the weighted average so that the location of merged clusters accurately reflects the items it represents.

Intersection details. A merge may introduce multiple intersecting pairs of clusters. Furthermore a merge can lead to new intersections because merging two child clusters creates a bigger cluster at a different location. We can solve this by defining an ordering beyond the zoom level of begin-to-intersect over intersections and merge intersecting clusters as long as there are intersections. The sub-ordering we use is largest area of overlap (as done in [25]), but this is one of many strategies. If the ordering still considers two intersecting pairs of clusters equal then we only have to make sure this is handled deterministically but the actual strategy used has almost no visible effect since it is very unlikely that this case occurs. If the ordering is not deterministic then the results of running the algorithm many times on the same input are not the same, in which case users have to re-establish or correct their mental map.

Algorithm. The algorithm works by starting at a zoom level at which none of the n clusters intersect (the initial zoom level) and then zooming out and merging clusters that thereby intersect (in order) until one cluster (the root) remains. One can keep zooming in until no clusters intersect. The zoom level of a collision between two clusters (when they should merge) can easily be computed in $O(1)$ time. We can make a $O(n^3)$ running time algorithm by computing at which

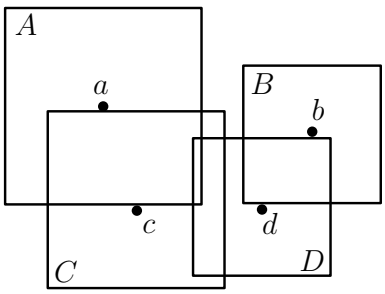


Figure 2.10: Four clusters represented by black dots a through d. Squares show the maximum size clusters can grow without intersecting others.

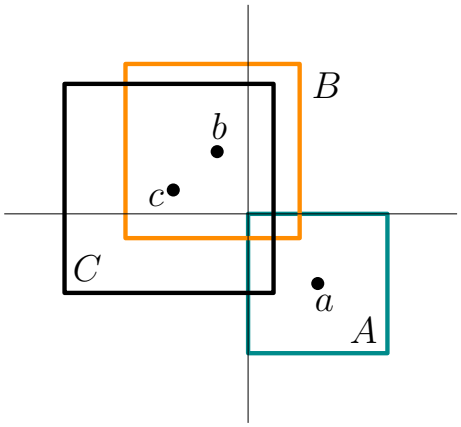


Figure 2.11: Squares A, B and C with centers a,b and c, respectively.

zoom level each of the $\binom{n}{2} = \theta(n^2)$ pairs of clusters intersect, merging the pair which intersects the earliest (at the highest zoom level) and repeating this another $O(n)$ times.

Figure 2.10 illustrates that we do not have to consider all $(1/2)n(n-1)$ pairs to find the pair of clusters that touches first as we zoom out. In this case there cannot be $\binom{4}{2} = 6$ pairs that will possibly have to be merged. Clusters A and B will never be the first that have to be merged because clusters grow equally fast horizontally and vertically and B will always be merged with D first. We claim there are at most $O(n)$ such pairs.

Lemma 2.4.1. *Suppose we have a set of axis-aligned squares S in which no square contains the center of another in its interior. Then no square intersects more than 12 squares of equal or greater size.*

Proof. Let A and B be two intersecting squares in S. Suppose B is at least as large as A, then B contains at least one corner of A either in its interior or on its boundary. For each corner of A we count the number of squares containing it (B is counted in one or two corners of A). We denote these counters by c_1, c_2, c_3 and c_4. We claim for all $i \in [1,4]$ that $c_i \leq 3$ leading to the bound we have to prove. We show this only for the top-left corner of a square (with counter c_1), since the proof for the other corners is analogous.

We divide the plane into four quadrants by drawing a horizontal and vertical line that both go through the top-left corner of A, as done in Figure 2.11. Suppose B is counted in c_1 (B contains the top-left corner of A) and its center b lies in the top-left quadrant. We claim that B is the only square of this kind since the contrary cannot be true: if another square C at least as large as A contains the top-left corner of A and its center c lies in the top-left quadrant, then either B contains c or C contains b. This contradicts with our assumption stating that no square contains the center of another square. Following the same reasoning, the top-right and the bottom-left corners can only add one to c_1. The bottom-right quadrant contributes zero to c_1 because any

square D with a center d contained in the bottom-right quadrant satisfies: either D contains a or A contains d. This again contradicts with our assumption. □

From Lemma 2.4.1 it follows that only $O(n)$ pairs have to be considered for the first merge. We can compute all nearest neighbor distances among n cluster centers in $O(n \log n)$ time [10]. Their k pairwise intersections can be computed in $O(n \log n + k)$ time with a simple sweep-line algorithm [10]. We know that $k = O(n)$ and since computing the zoom level of collision of each of k pairs takes $O(1)$ time we can compute the zoom level of the first merge in $O(n \log n)$ time. If we now repeat this procedure $O(n)$ times the final running time becomes $O(n^2 \log n)$. But we can still do better by reusing work. The nearest neighbor distances as well as the intersecting pairs barely change when a merge has been performed and so we compute them initially and update them after each merge using a dynamic nearest-neighbor graph (NNG) from [11]. Doing this yields an $O(n^2)$ time algorithm. We stress that n is the number of clusters (locations) and not the total number of items (for Trove 59 million books are published at 8000 different locations, so this is a big difference).

Computing when clusters should merge. Before we give the formula to solve we first recall the L^∞ (or *Chebyshev*) distance between two points (x_1, y_1) and (x_2, y_2) as being the maximum of $|x_1 - x_2|$ and $|y_1 - y_2|$. The unit "circle" in L^∞ is a square. Consequently we call half the side-length of a cluster its the radius.

The zoom level i at which two clusters x and y begin to intersect (touch) is

$$\begin{aligned} 2^{-i}(1.25^i(b_x + b_y + r_x + r_y)) &= c \\ (5/8)^i(b_x + b_y + r_x + r_y) &= c \end{aligned} \tag{2.1}$$

where

1. b_x and b_y are the border thicknesses of cluster x and y, respectively;
2. r_x and r_y are the radii of the interiors of the glyphs of x and y, respectively;
3. c is the L^∞ distance between the centers of cluster x and y.

Solving Equation 2.1 for i yields $i = \log_{5/8}(c/(b_x + b_y + r_x + r_y))$ which can be computed directly.

Note that this formula scales the sizes of the glyphs by 1.25^i. Zooming can change the fullness of the map quite drastically because clusters can split or merge and may seem to move. We try to remedy this by varying the screen-space radii of the clusters with zoom level, instead of keeping them fixed. We use a simple model, where all clusters are scaled by a factor of b^i. Here b is the multiplicative factor by which the clusters grow with each zoom step. For the Risse dataset we have $b = 1.25$.

2.4.1 Computing glyph sizes

The old GlamMap uses an accurate algorithm to compute the layout and size of a cluster representing m items (bibliographic records). A layout is the arrangement of tall rectangles, each representing an item in the cluster, in a grid. This grid is laid precisely on top of a rounded square

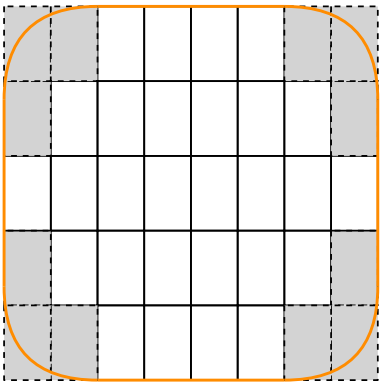
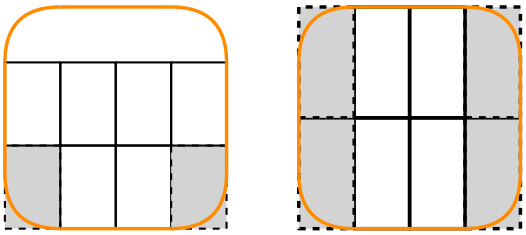


Figure 2.12: A layout of tall rectangles.



(a) Unadjusted grid (b) Adjusted grid

Figure 2.14: Two layouts of 8 tall rectangles.

(the interior of glyphs) as shown in Figure 2.12. For each cluster a number of different layouts (say k) is considered and the best one is chosen. A layout of 28 items in a grid with 5 rows and 8 columns can be seen. In each corner 3 tall rectangles of the square grid (highlighted in light gray) intersect rounded corners and they cannot be used since otherwise rectangles would be partially outside the glyph or overlap with its border. Note that this layout can also be used for any cluster representing less than 28 items, but space can be wasted. For example: if the layout of 28 tall rectangles is used for a cluster representing one item, then there obviously exists a smaller layout. Note that the size of tall rectangles is adjusted such that the grid outline is always a square, as illustrated in Figure 2.14.

Computing one layout. Computing the layout of a grid of m cells boils down to computing for each row the number of tall rectangles that do not intersect with rounded corners. If there is no such intersection then this is simply the number of columns of the grid. Otherwise, we define how we compute the number of columns for rows intersecting the bottom rounded corners. We do not define this for rows intersecting the top rounded corners as this is analogous. We know the following values:

1. Let w be the width of a tall rectangle.
2. Let t be the y-coordinate of the bottom of the row for which we are currently computing the number of columns.
3. Let r_1 be the radius of the circle of which one quadrant is the rounded corner.

We are looking for a value x, denoting the x-coordinate of the left of the narrowest part of the current row in between the bottom rounded corners. The relationship between these variables is illustrated in Figure 2.16. By math we have $x = r_1 - \sqrt{r_1^2 - t^2}$, which can be computed directly. The number of tall rectangles in the current row, intersecting bottom rounded corners, is $2\lceil (x-\varepsilon)/w \rceil$, where $\varepsilon = 0.01$ is a parameter that determines the tolerance for overlap between tall rectangles and the border (the effect can be seen in Figure 2.7, where some tall rectangles fall slightly outside of the glyph interior). For each of the $O(\sqrt{m})$ rows in a layout we spent $O(1)$ time thus a layout can be computed in $O(\sqrt{m})$ time. The representation of a layout is the number of columns for each row.

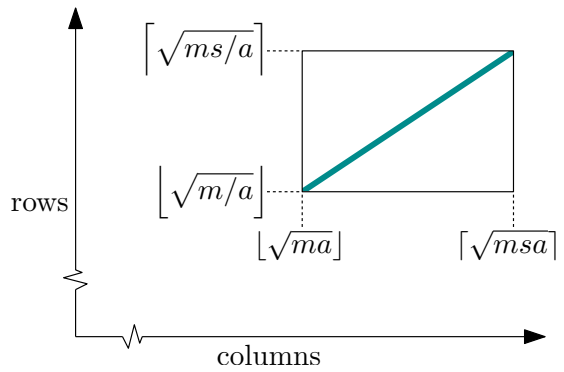


Figure 2.15: When choosing a layout for glyphs we consider a rectangle of arrangements (number of rows and columns) in the plane.

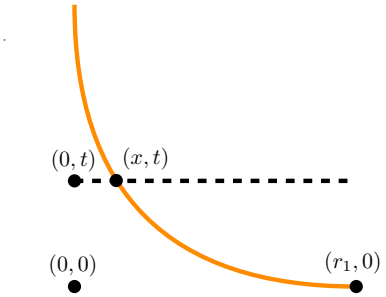


Figure 2.16: t is the y-coordinate of the bottom of the lowest row, r is and x is to be computed.

Which different layouts should we consider for one cluster? There are multiple layouts that can be used for one cluster. For this reason we consider numerous layouts and choose the best. Let s satisfy the equation $s = 1/((1-2r_2)(1+2r_2))$ where $r_2 \in [0,1/2]$ is a parameter that determines how round the corners are ($1/2$ is a circle and 0 is a rectangle). Our fixed value of $r_2 = 14/60$ yields approximately 1.28 (rounded upwards) for s. Let $a = 3/2$ be the desired aspect ratio for tall rectangles, each representing items. All layouts with a number of rows in $\left[\left\lfloor\sqrt{m/a}\right\rfloor, \left\lceil\sqrt{ms/a}\right\rceil\right]$ and columns in $[\lfloor\sqrt{ma}\rfloor, \lceil\sqrt{msa}\rceil]$ are considered. If each layout we consider was represented by a point in the plane, as illustrated in Figure 2.15, these points would form a rectangle. Not all layouts considered this way can actually contain (accommodate for) enough tall rectangles, so we simply discard those that are too small. Furthermore, layouts of which the first row will be empty are discarded since those waste too much space. The best layouts are those which are not discarded and have a tall rectangle aspect ratio closest to a. In Figure 2.15 these will be close to the green diagonal.

Running time. We consider k layouts and note that $k = O(m)$. For each layout we spend $O(\sqrt{m})$ time. The running time of this algorithm is $O(m^{1.5})$, assuming parameters a and s are constant.

2.5 Interaction and animation

The user interface of the old GlamMap uses an animation when zooming. It is important that GlamMap is fast and the mental map of users is preserved, which can be achieved by fluent interaction and animations. Interactive zooming is supported by rotating the mouse wheel, panning by anchoring the map to the mouse pointer while the left mouse button is pressed. Details on demand are supported through selecting clusters by clicking on them. The original GlamMap is not optimal in terms of responsiveness, since the clustering is computed client-side and takes several seconds. Furthermore, the rendering is rather inefficient since zooming may cause the web-browser to be unresponsive for up to one second.

Preview animation. To assist in preserving the mental map of users when zooming in, a preview of the next zoom-level can be seen. More precisely: while a glyph is pointed at by the mouse pointer, its child glyphs can be previewed (transparent at $5/8$ their normal size). Figure 2.17 shows this preview. When zooming in the child glyphs in the preview animate to their new locations. We know that when zooming from zoom level i to $i+1$ the width and height of the geographic region visible in the viewport shrink by a factor two. In other words: if the geographic region visible in the viewport is 100 kilometers wide at zoom level i, then at zoom level $i+1$ the visible region is 50 kilometers wide. In order for the size of a glyph to remain unchanged when zooming in, the glyph has to shrink by the same factor. Note this factor is not $1/2$ but $5/8$, as explained in Chapter 2.4. When animating previewed child glyphs into position their size does not have to be changed, because they were already shrunken by the same factor due to them being previewed. The locations can easily be animated since we just translate the original cluster centers to the new centers. Since the size of clusters is actually not expressed in the map coordinate system but rather in that of the viewport, resizing clusters is implicit.

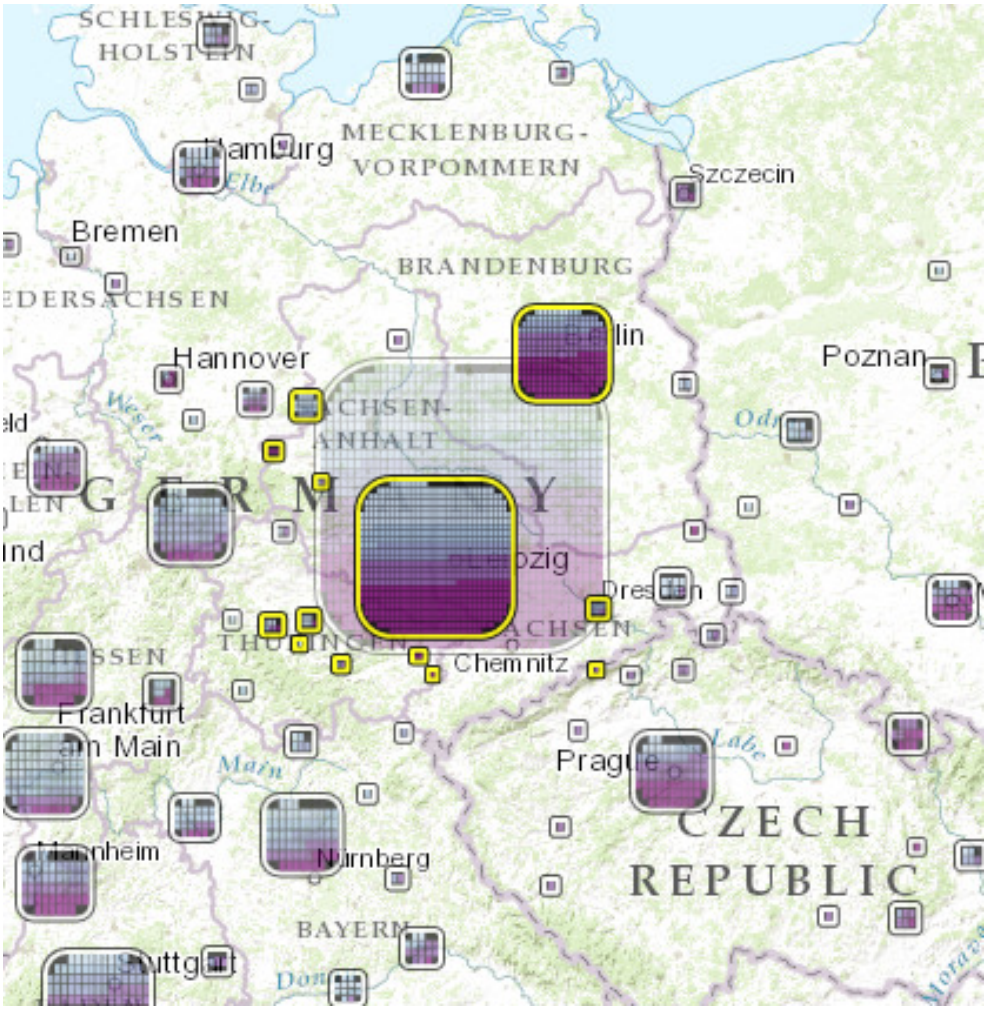


Figure 2.17: A preview of a cluster, showing that cluster's children.

2.6 Technical introduction

The old GlamMap is a standalone web-page, but a website properly utilizing the computing power of a server has not been fully realized yet. This prototype can be used without connection to the internet, if you do not consider that map tiles have to be downloaded from ArcGIS [1]. The clustering and glyphs overlaying the map have to be computed and rendered in the web-browser. GlamMap initially shows an overview of the entire dataset and since no server is used this means a list of all records has to be embedded in the prototype web-page. If any other datasets besides Risse have to be supported, they also have to be embedded in the web-page. This solution does not scale very well. As the number of datasets and their size increases, loading the web-page will take more and more time. For a prototype this is fine, but since showing the initial overview of the Risse dataset already takes several seconds, showing Trove (which is thousands of times larger) will take too long.

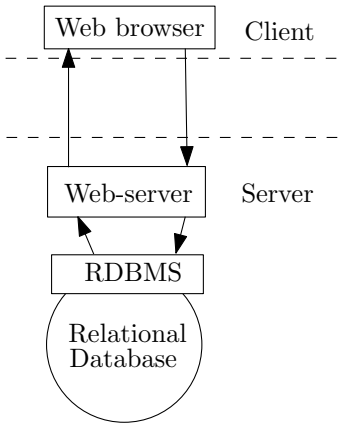


Figure 2.18: A data retrieval cycle in a client-server architecture using a relation database.

Current efforts towards a client-server architecture. Fortunately some efforts have been made to port this prototype into the better client-server architecture, but the only functionality fully ported is showing data in the details-on-demand panel. The clustering algorithm has been ported to C++ (except for the sub-algorithm computing the size of clusters) but is not yet integrated with the server. Data shown in the panels besides the detail-on-demand panel (the year and author panel) originates from the dataset embedded in the web-page, as opposed to being downloaded from a database on the server. Furthermore, any filtering of data and computing the clustering is still done client-side (on the user's device). Data shown in the details-on-demand panel is retrieved via the server from a *relational database* (an organization of data in interrelated tables). This data-access follows a cyclic pattern, as shown in Figure 2.18. Squares are software components and circles indicate significant storage (in this case only the relational database). The web-browser on the client, showing GlamMap, asks the web-server for data, which in turn forwards the request to the *relational database's management system* (RDBMS). The RDBMS is an interface for the relational database and actually retrieves the data. Thereafter the data is sent back to the client through the web-server. This cycle can enable a highly scalable application because transfers of large amount of data can be limited as much as possible. In the case of the detail-on-demand panel the filtering is applied as early of possible, at the RDBMS. If on the contrary filtering were to be applied somewhere else, e.g. at the web-server, then instead of just transferring the filtered data from the database to the web-server we would have to additionally transfer all unfiltered data.

Overview of technologies. An overview of physical servers involved directly in GlamMap is shown in Figure 2.19. One or more clients (the top row of the image) can use GlamMap (simultaneously). Each client uses the ArcGIS server to retrieve tile images that make up the map.

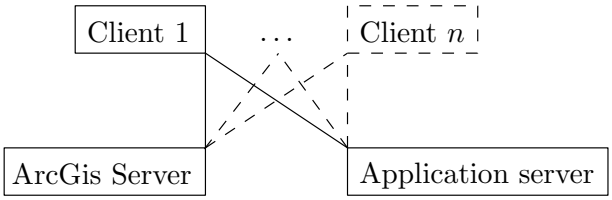


Figure 2.19: This figure shows which physical servers are used directly by the GlamMap application.

The 64-bit application server contains any GLAM datasets in its database and its role is to provide access to this data and clusterings thereof. The operating system that runs on the application server is CentOS 6.5. It in turn runs MySQL 5.6 (the RDBMS) and the web-server (Apache). The website is built using Ruby on Rails (ROR), a framework for websites, and is written in Ruby 1.9.3. The ROR component retrieves data from the RDBMS through an *Object Relation Mapping* (ORM) software library. ORM frameworks bridge the gap between representations in a relational database (tables) and in an object-oriented runtime (object graphs). Our ROR application forwards any clustering operations to a C++ component, which can compute them faster. Clients only require a web-browser, specifically Google Chrome without plug-ins, and GlamMap therefore runs on any platform that runs Google Chrome. The client side component of the web site uses HyperText Markup Language (HTML), Cascading Style Sheets (CSS) and Scalable Vector Graphics (SVG) to display information. Interactivity within GlamMap is implemented solely using JavaScript and a technique used to perform HTTP requests in scripts known as AJAX (Asynchronous JavaScript and XML). AJAX is used to retrieve data from the server without reloading the page and thereby allows little parts of the web-page to be updated without the overhead of loading the entire web-page. The client uses the Leaflet software library to show the map and other than that is fully custom built (it does not use any visualization toolkits).

The most significant specifications of the server are:

CPU — Intel Xeon E5-2420 1.90GHz

RAM — 32GB RDIMM 1600 MHz

Hard Drive — 3TB, Near-Line SAS 6.5Gbps 7.2k RPM

Any measurements were measured with this server machine.

Geocoding and GlamMap Geocoding is the process of translating location names to latitude longitude pairs. This can be as simple as searching in a list of name-coordinates pairs. One pair maps a city name to a latitude-longitude pair. A problem that can occur when geocoding is ambiguity of names. For example London most frequently refers to the place in the United Kingdoms but may also refer to a place in Canada. Most services that provide geocoding simply list all latitude-longitude pairs involved in the ambiguity and leave disambiguation to applications. Numerous organisations provide geocoding services. Google and GeoNames.org are examples of which the latter is used by GlamMap. Providers of free GeoCoding services limit the rate at which requests can be performed so that applications cannot overload their services. This rate-limit is enforced for each user (application). Applications are identified and authenticated through accounts.

GlamMap uses the geocoding service GeoNames.org, which geocodes based on English location names such as “New York” or “London” into coordinates on the spherical model of the

earth known as WGS84. The simple disambiguation scheme we use is: choose the first location obtained from the geocoding service. Consider the example of "University of Phoenix". In this case the first location returned is the stadium of the University of Phoenix. We erroneously choose this instead of the educational facility, thus this disambiguation scheme is not free of errors. GeoNames.org has a large list of place names (over 8 million) and has a rate-limit of 30 thousand location names per account per day.

Rendering glyphs. Clusterings are rendered client-side using an image-based technique. This technique draws glyphs to off-screen buffers, which are then converted to images and rendered by the browser. Whenever zooming, panning or filtering occurs we *invalidate* the visualization, causing three images for each glyph in the entire hierarchical clustering to be redrawn. Each of these three images has a different inner border colour, serving as feedback for the user when hovering the mouse pointer of a glyph or clicking on a glyph. When the web-page loads we invalidate the visualization. In addition, the first time the mouse pointer is placed over a glyph with children after the last invalidate, we draw an image for each child cluster (to show the mouse-over preview).

2.6.1 Database

GlamMap is capable of storing any bibliographic metadata and represents it naturally and intuitively by using a relational database, which structures data in interrelated tables, with a design based on Functional Requirements for Bibliographic Records (FRBR) [21]. The RDBMS (we use MySQL Server 5.6) stores data in tables, ensures any constraints over this data are satisfied and enables data-queries to be answered quickly.

FRBR influence. FRBR defines concepts commonly found in bibliographic datasets and more generally any kind of collection of cultural manifestations. We use FRBR as guidelines for our database to make it more compatible with any GLAM datasets to be added in the future. The main concept of FRBR we use is its four-level hierarchy (the FRBR hierarchy) consisting of *works*, *expressions*, *manifestations* and *items*. A work is a distinct intellectual or artistic creation and is an abstract entity (e.g. Ronald Haymans Playback) and has many expressions. An expression is the intellectual or artistic realization of a work (e.g. the author's text, edited for publication) and has many manifestations. A manifestation is physical embodiment of an expression or work (e.g. the book published in 1973 by Davis-Poynter) and has many items. An item is a single exemplar of a manifestation (e.g. copy autographed by the author).

Formal definitions. We refer to the following definitions in the remainder of this section:

1. Let t be a table, then $C(t)$ denotes the set of columns of t.
2. Let t be a table, then a *row* r of t is a tuple with one element for each column in $C(t)$.
3. Let t be a table, $C' \subseteq C(t)$ an ordered set of columns of t and r a row of t then we define r *under* C' as r without elements for columns $C(t) \setminus C'$.
4. Let c_1 and c_2 be columns and let D_1 and D_2 denote the universe of values that rows can take under columns c_1 and c_2, respectively. We say c_1 is *compatible* with c_2 if $D_2 \subseteq D_1$.

5. Let C_1 and C_2 be ordered sets of columns, then C_1 is *compatible* with C_2 if $|C_1| = |C_2|$ and for all $i \in [0, |C_1|]$ the i'th column in C_1 is compatible with the i'th column in C_2.
6. Let t be a table, then a *unique constraint* over columns $C' \subseteq C(t)$ requires that no two rows of t can have the same values under all columns in C'.
7. Any table t has at most one *primary key*, an ordered set of n columns $PK(t) \subseteq C(t)$, which by default imposes a unique constraint (also known as the *primary key constraint*) over $PK(t)$.
8. Let t_1 and t_2 be tables and $FK \subseteq C(t_1)$ an ordered set of columns compatible with $PK(t_2)$, then a *foreign key constraint* over columns FK referencing t_2 requires that for any row of t_1 with values V under columns FK there exists a row of t_2 with values under $PK(t_2)$ equivalent to V. We say FK is a *foreign key* referencing t_2.

Structure and constraints. The structure of the database of GlamMap contains the FRBR hierarchy and is shown in Figure 2.20. Here a *database schema* can be seen, which is a definition of tables and their columns and relationships between these columns. The primary keys of a table are highlighted in blue. Foreign keys are indicated by arrows pointing to the referenced side. Note that foreign keys are not defined in the database in GlamMap, but rather are implicit (this means that the application GlamMap assumes these foreign key constraints hold). Furthermore, this is a simplification of the actual database schema (e.g. libraries, their locations and holdings are not included), but shows all relevant information. Short descriptions for each table are:

`dataset` This table contains metadata for each dataset, such as their names. A dataset has many works.

`dataset_work` This table realizes the many-to-many relationship between works and datasets.

`expr_person` This table associated people with expressions. The column `role` indicates in which role a person is associated with an expression (e.g. author of a book or painter of a painting). If the primary key would only contain the columns `role` and `expr_id` then multiple people cannot be associated with an expression in the same role (e.g. an expression cannot have multiple authors). For this reason we add an additional column `prio`, which stands for priority, to the primary key. This column represents the relative importance of persons associated in the same role with one expression.

`expression` Each row represents a FRBR expression and can be associated with many people.

`item` Each row represents a FRBR item.

`loc_name` Contains location names. A unique constraint is imposed on `name`.

`location` Each row represents a different location and therefore two different locations may not have the same geographical coordinates. A location has one or more names.

`mani_loc` Associated locations with manifestations. Each association between a location and a manifestation must have a different type (e.g. the location of publishing of a book).

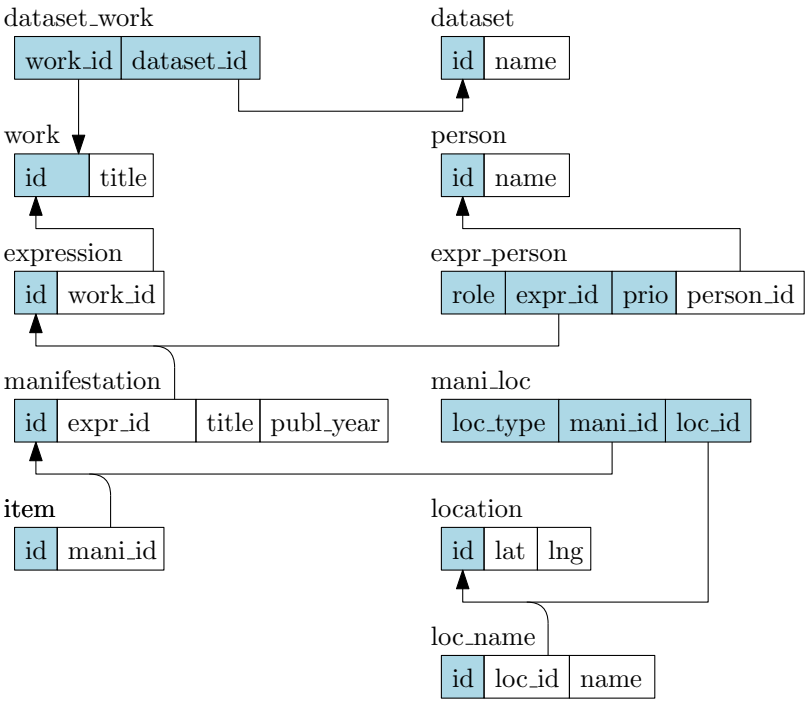


Figure 2.20: The schema of the database used in GlamMap.

`manifestation` Each row represents a FRBR manifestation and can be associated with many locations (each location must have a different type). The column `publ_year` is the year in which this manifestation was published (unveiled). We require that each manifestation is associated with a location of publication (otherwise it cannot be shown on a geographical map).

`person` Contains the names of people. Each person can be associated with many expressions. A unique constraint is imposed on `name`.

`work` Each row represents a FRBR work.

The following data types are used:

1. All `name` and `title` columns use the data type `VARCHAR(255)`, a small sequence of character (text).

2. The columns `lat` and `lng` use the numerical data type `DECIMAL(10,6)` (the integral and fractional part have at most 4 and 6 digits, respectively).

3. Any other columns use the integral data type `INT` (a four-byte singed integer using two's complement representation).

Search-indices. A *search-index* is a data structure that allows rows to be found and queries to be answered quickly, but requires extra storage space and maintaining it over insertion, deletes and

updates of rows costs additional time. The data structures used for search-indices and their exact implementation depend on the RDBMS that is used. MySQL 5.6 forwards this responsibility to one of many pluggable *storage engines* of the RDBMS (defined on a per-table basis). The storage engine of a table defines the format of this table, implements search-indexes on this table and executes SQL commands. The default and general-purpose storage engine used by MySQL 5.6 is InnoDB, which implements all search-indices (except full text and spatial data indices) using *B-trees* (we strictly use InnoDB in GlamMap). A B-tree is a generalization of a binary search tree where nodes can have many children, which makes B-trees shallower than binary trees, and is widely used in relational databases. In InnoDB a full text index is similar to an inverted index and indices over spatial data use R-trees. A B-tree on a table t is defined with respect to a key $K \subseteq C(t)$ (keys are ordered sets of columns) and leaves either point to rows or contain the rows themselves. The in-order tree walk of this B-tree will visit rows contained in (or pointed to by) leaves as if t was sorted lexicographically on K. There need not be a unique constraint over K and therefore B-trees support leaf rows with the same value under K.

Range queries on B-trees. A B-tree on table t with key $K \subseteq C(t)$ can be used to efficiently find ranges of rows (as opposed to hash tables, since they have no order). The *size* of a key K is $|K|$. If there is no unique constraint over K then searching in this B-tree using a specific value for K may already produce multiple rows (since different rows may have the same values under K), but we can also find multiple rows (ranges) by searching for *prefixes* of keys. If $C' = (c_1, c_2, \ldots, c_n)$ is an ordered set of n columns then a *prefix* of size $k \leq n$ of C' is the set of columns $(c_1, c_2, \ldots, c_k)$. We say a prefix is *strict* if $k < n$. In the extent of finding ranges using prefixes, consider a B-tree on table t with key $K \subseteq C(t)$ such that $1 < |K|$ and let K' be a strict prefix of K. We want to find the range of rows with value v under K'. Since B-trees order their rows, this is equivalent to finding the first and last row of this range while reporting every row in between. To this extent let $v_{\min}$ and $v_{\max}$ be the smallest and largest, respectively, values that any row r can have under K such that r has value v under K'. We can proceed like a normal search for both $v_{\min}$ and $v_{\max}$ (preferably not traversing their common path starting at the root twice) to find the first and last row we have to output, respectively. Finally, any subtrees in between both search paths have to be traversed in the appropriate order, so that any nodes between the first and last row can be reported.

Alternatively, a variation of B-trees can be used that adds pointers to all leaf nodes to make a linked list of all leaf nodes (e.g. B+ trees). Of course this linked-list should respect the order induced by K to be easily usable for range queries. In this case we start at the linked-list/leaf node containing (or pointing to) the first row and iterate over successive nodes until the linked-list/leaf node containing the last row is encountered (here we do not have to traverse the tree to find the node containing the last row).

Search-indices and primary keys. A database implementation often creates a search-index for each table t with a primary key by default (the *primary key index*), to support the common operation of finding the row with a particular *identifier* (value of this row under $PK(t)$). If the primary key index is implemented using a B-tree, then typically leaves of this B-tree contain rows themselves rather than pointers to these rows. We then call the primary key index a *clustered index*, since the rows on disk are now clustered (sorted by $PK(t)$). More so, if table t has a

primary key, pointers to rows are typically just their identifiers. MySQL 5.6 also creates search-indices for primary keys, but this is not guaranteed to be a B-tree (by documentation of MySQL).

Foreign keys. MySQL 5.6 requires for all tables t (we assume t uses the InnoDB storage engine) for all foreign keys FK on t (referencing any table) that a B-tree i on t with key K exists such that FK is a prefix of K. MySQL requires when deleting rows in referenced tables that any references can be found quickly (to verify consistency) and to this extent demands a *supporting* index (i). Note that if FK references t itself the supporting index can be the primary key index and may not be a B-tree. When defining foreign keys in MySQL 5.6 on tables using the InnoDB storage engine, a B-tree index with key FK is created if no supporting index exists. MySQL has a slightly more generic definition of foreign keys and does not require that the ordered set of columns $C(t_2)$ of the referenced table t_2 is the primary key, but only a B-tree index on t_2 with key $C(t_2)$ should exist (there may be multiple referenced rows for one referencing row).

Tuning. MySQL 5.6 provides many settings to increase the throughput of data-queries and data mutation operations. GlamMap initially uses the default settings. Choosing good settings depends on the level of concurrency (how many users are using the database at the same time) and server capabilities. GlamMap does not have a high level of concurrency, but does not fully utilize the server capabilities. Consider for example one of the most important settings, named `innodb_buffer_pool_size`. The default value for this setting is $1/8$GB, while the server has 32GB of memory. This setting serves as an in-memory cache for search-indexes and table data stored on disk (for those tables using the InnoDB storage engine). This means that we only read search-index and table data from disk if it is not in the cache. Furthermore, we do not have to write changes of search-index and table data to disk if they are in the cache. If enough memory is available then entire tables and search-indices can be cached. Since a disk is much slower than random access memory (RAM) this can greatly increase performance.

Partioning. MySQL 5.6 allows tables to be partitioned. As the name suggests this partitions tables into several subtables. A search-index on this table is scoped to each partition (we have one search-index per partition). The partitioning function determines which row is assigned to which partition. Consider a table that stores for books their year of publishing. We can partition this table into partitions p_1 and p_2. Books published before the year 1950 are stored in p_1 and the remaining books are stored in p_2. Suppose furthermore that a B-tree search-index over the year of publishing has been defined (each partition has its own B-tree). When querying for books published in a particular year range, the query optimizer can now prune partitions if no rows can be found in them (before any index or table data has to be read). If this table was not partitioned but retains its index then this B-tree search-index would be deeper (since it is defined over more rows). In fact: the sum of the size of indexes on partitions is less than the size of an index over the entire table. If we perform range queries on the unpartitioned table then pruning is solely done by traversing the B-tree (requiring index data), whereas pruning can be done by excluding partitions (at the query optimizer) without reading any index data. For these reasons using partitions for range queries may be faster and require less storage than other partitions. MySQL allows queries to be restricted to particular partitions.

Parallelism. MySQL 5.6 does not support executing (parts of) one query in parallel, but only

allows different queries to be executed in parallel. For GlamMap the latter is not significant, since we do not have a lot of users. The clustering-based overview shown in GlamMap represents each row in the dataset thus data-queries for the input of this overview must summarize potentially very large datasets. For Trove we have to be able to filter and aggregate all 60 million records. If we assume the relevant part of the Trove dataset fits in RAM, then finding the clustering input is CPU-bound and we do desire parallel query execution. This is especially true if one considers that CPU architectures of today and tomorrow focus more on horizontal (more cores and parallelism) than on vertical scaling (higher CPU frequency).

Chapter 3

Integrating Trove with GlamMap

This chapter describes how the GlamMap prototype was upgraded to realize a proper client-server architecture and is capable of visualizing Trove [4] (60 million records). Up until now the GlamMap prototype had only been used to visualize small datasets of at most 8000 bibliographic records, in which case little or no scalability issues arise. Before we can attempt to run the clustering algorithm on the Trove dataset and visualize it, we first have to retrieve it from a service provided by the National Library of Australia, store it in the relational database and geocode it using GeoNames.org. These three steps together are called *importing Trove*. After these steps we still need to be able to access and filter the data efficiently, compute clusterings, and render them. The problems that arose while integrating GlamMap with Trove are:

1. Downloading Trove from its source, converting and Geocoding it takes a week to complete and abruptly stopping this task due to an unforeseen error should not lead to work being lost, but rather software should be resilient and capable of continuing where it stopped to save time;
2. Due to the flexible format of Trove many different patterns can be found in the data and are used to extract this data into a simpler common format, but checking for many different patterns will lead to this process taking days instead of hours;
3. Geocoding is difficult because the geocoding service we use only accepts concise location descriptions (at most two words) and does not tolerate spelling errors We often have long location descriptions (three or more words) with spelling errors.
4. Retrieving the data for the clustering algorithm and panels is time-consuming and these queries will not terminate within hours unless the database is optimized, but we require it to finish within several seconds.
5. Running the clustering algorithm itself, in particular computing the size of glyphs, takes several hours due to a very accurate but slow algorithm. Marginally reducing the accuracy of the algorithm while gaining time solves this problem.
6. Drawing the visualization on the end-user's device (client) makes the web-browser unresponsive for seconds, but when drawing only what is visible this process finishes quicker.

3.1 Importing Trove

The dataset of Trove can be retrieved from a public Application Programming Interface (API), the *Trove-service*, provided by the National Library of Australia. From this service we can download all 60 million (roughly) books records. We do not count *holdings* (a holding is a copy of a book available at a library) of these books, but only the books (manifestations in FRBR) themselves. The format (representation) used by the Trove-service is then converted into our FRBR representation and geocoding is performed. Integrating Trove with GlamMap has been envisioned for a long time and we used an existing snapshot of the books in Trove (obtained from the Trove-service), made in November 2014. This snapshot does not include holdings.

API details. The URL and its parameters used for enumerating all books in Trove, exemplified for one page, are shown below.

```
http://api.trove.nla.gov.au/result?
    q=format:book & zone=book & include=workversions &
    l-year=1841   & n=100     & s=15800
```

The first two parameters specify that we are interested in books. These parameters may seem redundant but are both mandatory. The third parameter specifies that we are not just interested in works, but also in their contained versions and Trove-records. If we additionally want holdings, this parameter will look like `include=workversions, holdings`. The last three parameters control which page of works we want and in fact we have two dimensions of paging. The first dimension is the year and is controlled by the fourth parameter. The second dimension is the window and is controlled by the last two parameters (`n` is the size and `s` is the offset of this window). Note that the year parameter is just a filter on a particular year and one might ask why such a complicated paging scheme is needed instead of just using the window (removing the fourth parameter). Since the snapshot was already created, experimentation with different paging schemes is outside the scope of this thesis, but the more complicated scheme may be faster.

Trove format. Trove data is structured according to the *Trove-hierarchy*. This hierarchy is similar to that of FRBR, but only has three levels instead of four. The nodes in the hierarchy are work, *version*, *Trove-record* and holding, of which the latter two are on the same level. In fact we have the following relationships:

1. A work has one or more versions.
2. A version has one or more Trove-records.
3. A version has one or more holdings.

An additional relationship was mined from the snapshot, and may not hold for Trove in general, but does hold for the subset of books of Trove. We found that each version has at most one Trove-record. This hypothesis was first thought of when inspecting the snapshot to learn about Trove's format and could be verified programmatically for the entire book snapshot. The relationship between versions and records therefore is one-to-one and a version and its related Trove-record can be interpreted as one entity.

The Trove format used by the Trove-service is a very elaborate one and contains much information we do not need. Consider for example the work shown in Listing 3.1. The `url` and `troveUrl` of a work can be derived from its `id`. The first and second version of this work represent a book and an article, respectively. We explicitly asked for books, but this only affects works and not their contained versions and Trove-records, which explains why an article version is present. The `issued` seems to show the minimum and maximum years during which contained versions and Trove-records were issued. For the second version this year even is incorrect and this error propagates to the containing work (we do not correct this error and hence we still have artefacts in our data). The first version and its related Trove-record are associated with their corresponding record in OCLS's WorldCat [24] database through an `identifier`. Fields associating Trove-records with other databases (other `identifiers`) and the sources of this metadata have been removed in this example.

The field `publisher` is the only field containing hints of precise geographic locations and `creator` seems to contain author names. A Trove-record can have zero or more `creators` and one or more `publishers`. These fields are text-based and seem to follow a wide variety of patterns. Consider for example `creator` samples below (obtained from the snapshot):

1. `McTaggart, D.`
2. `Rarey Mr.`
3. `Religious Tract Society.`
4. `Brontë, Charlotte, 1816-1855. Jane Eyre.`
5. `Hugo, Victor, 1802-1885. Notre-Dame de Paris.`
6. `Farrar, F. W. (Frederic William), 1831-1903.`
7. `Wills, Freeman.`
8. `Martineau, Harriet, 1802-1876.`
9. `Martineau, Harriet, 1802-1876. Loom and the lugger.`
10. `Raymond, R. J. (Richard John)`
11. `Raymond, Richard John. Cherry bounce.`

In general we first have the surname, followed by (initials of) first names. Exceptions to this pattern are samples 2 and 3. Sample 3 seems to denote an organization rather than a person's name. Sample 2 seems to only show one name with a title. Samples 6 and 10 show full first names as well as initials. After these names a lifespan and little phrase optionally follow. Some Trove-records do not have `creators`, but author names may be present in `titles` and `publishers`.

```
<work id="26265603" url="/work/26265603">
  <troveUrl>http://trove.nla.gov.au/work/26265603</troveUrl>
  <title>The changing Australian home</title>
  <contributor>Dickinson, Michael</contributor>
  <issued>8-1981</issued>
  <type>Book</type>
  <isPartOf>National Times</isPartOf>
  <version id="31637464">
    <record>
      <title>The changing Australian home</title>
      <creator>Dickinson, Michael.</creator>
      <issued>1981</issued>
      <publisher>[Sydney] : National Times,</publisher>
      <identifier type="control number" source="OCoLC">219957412<
         /identifier>
    </record>
    <type>Book</type>
    <issued>1981</issued>
  </version>
  <version id="52067755">
    <record>
      <identifier type="rmitDocumentNumber">81063524</identifier>
      <title>The changing Australian home</title>
      <creator>Dickinson, Michael</creator>
      <isPartOf type="publication">National Times</isPartOf>
      <bibliographicCitation type="issue">8-14 Feb
        1981</bibliographicCitation>
      <type>Journal Article</type>
      <subject>Architecture</subject>
    </record>
    <type>Article</type>
    <issued>8-14</issued>
  </version>
</work>
```

Listing 3.1: An example of a work in Trove's XML-based representation.

Mapping of Trove to our database. The Trove-hierarchy is mapped to our database, which uses the FRBR hierarchy, as illustrated in Figure 3.1. The concepts of a Trove work and a FRBR work coincide nicely. Similarly, holdings and items both represent physical copies. A version is associated with exactly one Trove-record and therefore such pairs can be interpreted as one entity. For this reason we create one expression and manifestation for each Trove-record. Individual records are mapped as follows:

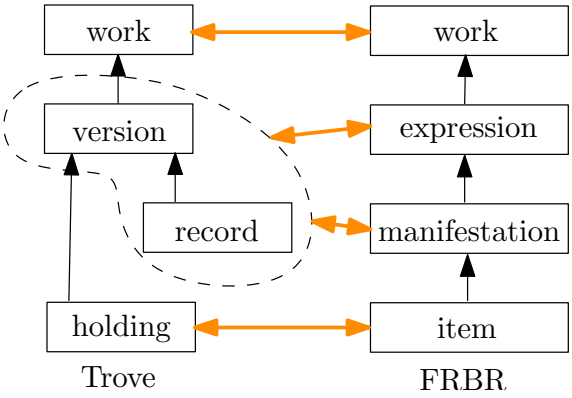


Figure 3.1: Mapping of the Trove-hierarchy to the FRBR hierarchy.

1. A FRBR work is created solely based on a Trove work, since we only need a title, if we decide to create at least one expression.

2. An expression is created from a version and Trove-record. No mapping is defined for versions not of `type` book (we do not create an expression for these). Additionally, if we decide to not create a manifestation from this version and Trove-record then we also do not create this expression. The first `creator` field, if any, of a Trove-record is interpreted to contain the main author of the represented book. Therefore we create a person with this name if it does not exist already and the person with this name is associated with the expression by creating a row in the `expr_person` table (with `role` creator and `prio` 10). We do not store multiple authors per book. This is not supported on the web-page and is not as important as showing a high-level overview, making it a time-intensive and lower-value task. Besides the main author, no other people are associated with the expression so the value for `prio` does not matter (the magic number 10 could be any number).

3. A manifestation is created from a version and Trove-record. If we decide to not create an expression from this version and Trove-record then we also do not create this manifestation. We interpret the `issued` field of a Trove-record as the year in which the represented book was published. This constitutes the `publ_year` of a manifestation. The `title` field of a Trove-record is mapped to the corresponding field in a manifestation. We use `publisher` fields in a Trove-record to determine the name of the location at which the represented book was published (more detail in Section 3.2). A location is created if one with this name does not exist already. Names are simplified first and we may decide a name does not denote any location, in which case we do not create this manifestation (see Section 3.2). This location is associated with the manifestation by creating a row in the `mani_loc` table (with `loc_type` published).

4. An item should be created for each holding, but we do not include holdings. Only one item is created for each manifestation so that the FRBR hierarchy is complete. No information is stored with this item, but would be associated with a library at the appropriate location if holdings were included.

Implementation and optimizations. The first implementation of the script importing Trove (not including geocoding) was written using the ORM library, which allows programmers to work quicker at the cost of a larger running time. Object Relation Mapping (ORM) takes care of the conversion between table representations in the database and object graphs in Ruby (or object-oriented runtimes in general). The ORM library thereby makes inserting and updating objects in the database easy and interfaces naturally with Ruby. Likewise, querying the database is easy. There are pitfalls however, which make this implementation of writing data to the database slow:

1. The amount of SQL (Structured Query Language) commands sent to the database (inserting, updating, deleting and selecting rows) should be limited since each command incurs overhead (e.g. parsing SQL and determining a strategy for the execution of a command). The first implementation did not group insert commands together (batch), incurring overhead for each single-insert command rather than for each batch. Furthermore, some rows were first inserted and later updated (two commands), while they could just as well be inserted in one command and not updated.
2. When creating objects in the database they are first validated by the ORM library, checking if all fields are defined on objects and the name of a person or location is unique within their respective tables. If this is not the case then the database will become inconsistent and therefore an error is produced. However, we know all fields are defined and furthermore we create a person or location only if one with that name does not exist already. If we assume persons and locations are not inserted concurrently then all checks are in fact unnecessary.
3. The ORM library requires that we provide object representations of rows to be inserted in the database. These are later converted to SQL commands and this additional translation can be avoided by creating SQL commands without help of the ORM library and directly embedding rows in this command.
4. It is rather costly to check whether a person or location with a specific name already exists, since we have to query the database for this, which in turn needs to perform a search in the person or `loc_name` table. If we assume that persons and locations are not updated or deleted at all then we can use the result of previous queries (thereby only checking if each name exists once rather than every time).

Our second implementation deals with all four pitfalls (exploiting the assumption that persons and locations are not inserted concurrently or updated and deleted at all), which is two-thousand times faster than the first implementation. To tackle (1) we merge any insert/update commands on the same row into one command (where possible) and stop using the ORM library since it does not support batch insertion. We instead create SQL commands directly, allowing us to tackle (2) and (3) as well. For (4) we implement caches by using a hash table for locations and another one for persons. Instead of querying the database as to whether a person or location already exists, we first query the respective cache. If a location or person is found we immediately retrieve the identifier of the location or person from the respective cache. The second implementation is a lot more memory-intensive, since large buffers are needed for batch commands and caches require a lot of space. In fact: if the batch size is 50000 we run out of memory on a Windows system

with the default Ruby settings (the memory limit is determined by the ruby runtime rather than the physically available memory), so we use a smaller value of 10000.

This script still takes 6 hours to complete and any errors will cause work to be lost, we track which part of the snapshot has been successfully imported and allow the script to start at a specified point in the snapshot. This way we can recover from errors without losing more than a minute of time.

3.2 Geocoding Trove

The geocoder initially used by GlamMap does not scale well to Trove (its rate limit is 30 thousand location names per day per account), has a simple disambiguation scheme and location names from Trove-records cannot be fed into the geocoder directly due to GeoNames.org not being invariant to artefacts. To this extent we scale the geocoder and simplify location names first (simplifying includes removing artefacts). Another problem of the initial GlamMap is that no duplicate locations are detected and merged if they have the same geographic coordinates but not the same simplified name (locations with the same name are dealt with earlier, as discussed in Section 3.1). Detecting and merging these is important because the clustering algorithm requires that two initial clusters do not have the same location. If we geocode all 60 million records separately in Trove, geocoding the entire dataset would take years to complete.

Processing location names. The main geographical location hints of Trove-records can be found in their `publisher` fields, which indicate where Trove-records (books) are published. A Trove-record may contain more than one `publisher` and since we only support one location of publication we simply only look at the first `publisher`. Few books seem to have multiple publish locations and our database only supports one such location per book, so finding and implementing any other way of dealing with this seems like a time-intensive but low-value task. Samples of `publishers` (from the snapshot) are:

1. `Leipzig : Reclam,`
2. `Paris : Garnier Freres,`
3. `[London : Cassell, Petter, Galpin,`
4. `Brisbane : Govt. Printer,`
5. `London : Milner,`
6. `[Japan,`
7. `Edinburgh : printed by Ballantyne, Hanson,`
8. `London : Stevens,`
9. `Dublin : A.M. Sullivan,`
10. `London : Religious Tract Society,`
11. `Boston, S.G. Simpkins,`

12. `Boston : Samuel G. Simpkins,`
13. `S.l : s.n,`
14. `London : [s.n,`
15. `Oxford : Oxford University Press,`
16. `[n.p.] : [n.p.],`
17. `Hartford [Conn.] : J. Holbrook, printer,`

This field seems to start with a place name, which we can use. Unfortunately these can contain spelling errors, non-alphabetical characters and names can be written in different languages or can seem nonsensical (e.g. `[n.p.] : [n.p.],`). The phrase `s.l.`, which is Latin for *sine loco* and means "without a place", is likely to indicate that the location of publishing for a book is not known or not specified. The general patterns this field follows are undocumented and can only be found through mining. To this extent we count the frequency of all unique location names (before simplification). One of eight-thousand locations are used in approximately 95% of all records and their location names follow a general pattern very similar to the samples above. We construct a small set of *sanitization-rules*, focusing on the most frequent general patterns. A *sanitization-rule* can be applied to a name (phrase) to simplify it or concludes the name does not denote any location. The sanitization rules we apply recursively (until no more can be applied or we conclude the name does not denote any location) are:

1. If there is a semicolon or colon in the name, replace the name by the part of the name before the first semicolon or colon (use only the first component of the name).
2. Remove accents from English alphabet characters.
3. Remove non-alphabetical characters in the set `[]()?` from the name.
4. Remove any leading or trailing spaces.
5. If the name starts with one or more dot-delimited letters (initials) then conclude there is no location (e.g. dispose of any person names written with initials and the literal `s.l.`).

Because we focus on the most frequent names and have a small set of rules, we have made a reasonable speed-accuracy trade-off (the more refined the set of rules is, the more accurate the results and the more rules need to be checked for applicability). The number of unique simplified location names in Trove is now four-hundred-thousand.

Geocoder changes. After processing location names we still need two weeks to geocode Trove, so we extend the geocoder to use an array of different accounts and track the rate-limit of each. By adding 7 additional accounts the capacity of the geocoder increases by a factor of 8. Note that the changed implementation still performs all geocoding requests sequentially and unless requests are performed in parallel this method stops scaling at 12 to 13 accounts. Geocoding the books in Trove can now be done in less than two days.

Post-processing. After simplyfing and geocoding location names we are left with locations with different names but the same geographical coordinates and names for which GeoNames.org did not give any geographical coordinates.

We remove the latter locations and since each manifestation (book) must be associated with a location of publication (all locations we import are places where books are published) we also remove any manifestations associated with removed locations. This removal also propagates to the sole expression associated with a manifestation and if a work does not have any more expressions it is also removed. This results in the removal of less than two (out of sixty) million books.

We additionally collapse any list of locations with duplicate latitude-longitude pairs to one location, by making any references to these locations point to the first location and removing non-first locations (in the list). We remain with 10000 unique places. Note that 95% of the books in Trove have been published at 8000 locations.

Ambiguities. We do not have a solution for the problem of ambiguity, where one location name may denote more than one actual locations. We retain the disambiguation scheme used initially by GlamMap: simply take the first location provided by GeoNames.org.

3.3 Optimizing the database for data-access

Several data-queries are performed while using GlamMap to show Trove and all of them do not terminate within one hour, thus massive optimizations are required to achieve interactive speeds. We retain the current storage architecture (the RDBMS known as MySQL 5.6) and focus on using it more effectively. Switching to a completely different RDBMS requires porting GlamMap and installing and configuring the RDBMS. This incurs overhead and we assume retaining the current RDBMS has the best result/time trade-off.

Summary of filters. The list of filters that are used throughout GlamMap is:

`publ_year_min` and `publ_year_max` are the minimum and maximum year, respectively, at which manifestations to select were published. They default to $-\infty$ and ∞, respectively.

`excluded_creator_ids` is a set of person identifiers. Only select expressions which are not associated in the role of creator (e.g. author of a book) with a person identified by this set. This defaults to the empty set.

`title_inclusion` is the text (sequence of characters) which must exist as a subsequence in the title of manifestations to select. This defaults to the empty sequence.

`publ_loc_ids` is the set of location identifiers of locations with which manifestations to select must be associated (only associations of type publication count). Defaults to the set of all location identifiers (this does not mean that the actual representation is a set of all location identifiers).

Queries to support. We list the data-queries we are dealing with and note that they are scoped to the currently selected dataset. When discussing queries for a filtered set of items we assume that the FRBR hierarchy is flattened, meaning for any item i we have know its manifestation m, the expression e of m and the work of e. We make a similar assumption when discussing a query

for a filtered set of manifestations: for any manifestation m we know its expression e and the work of e. The queries we require, for each user interface component, are:

- q_1 (clustering on map): the number of items selected by filters `publ_year_min`, `publ_year_max`, `excluded_creator_ids` and `title_inclusion` per location of publication, per year of unveiling category (bucket).
- q_2 (year histogram): the number of items per year of unveiling.
- q_3 (creator panel): let M be the set of manifestations selected by filters `publ_year_min`, `publ_year_max` and `title_inclusion`, for each creator of an expression in M the number of manifestation he or she created.
- q_4 (details-on-demand panel): a page of the set of manifestations selected by filters `publ_year_min`, `publ_year_max`, `excluded_creator_ids` `title_inclusion` and `publ_loc_ids`.

For queries q_1 and q_2 we require the lowest level of the FRBR hierarchy (items) and need to scope these items to the current dataset. Furthermore, for each item i we need its manifestation m and we need the expression e of every m, to apply filters. Scoping items to the current dataset can only be done through the works of expressions and the table associating works with datasets. Suppose the current dataset has identifier 5. The SQL query then joining the FRBR and dataset-scoping tables is shown in Listing 3.2. Note that we require similar SQL for queries q_3 and q_4, the only difference being we do not need items.

```
SELECT *
FROM work
  INNER JOIN expression ON expression.work_id = work.id
  INNER JOIN manifestation ON
    manifestation.expr_id = expression.id
  INNER JOIN item ON item.mani_id = manifestation.id
  INNER JOIN work_dataset ON work_dataset.work_id = work.id
WHERE dataset_id = 5
```

Listing 3.2: A SQL query joining FRBR scoped to the dataset with identifier 5.

Query q_1 almost directly yields input for the clustering algorithm, but it does not contain geographical coordinates. The clusters returned by q_1 therefore are first augmented with these coordinates before they are fed to the clustering algorithm.

General restructuring and optimization of q_2. The common part of queries q_1 and q_2 is shown in Listing 3.2. This join query takes longer than an hour to complete. We cannot optimize it by adding search-indices, since indexes are already defined on all columns used in the join and where clauses, therefore the RDBMS already considers all possible indices for this schema when constructing an execution plan. One thing we can do, however, is restructure the database. Since search-indices can only be defined over columns of one table, by performing joins on the schema level, we open up new opportunities for search-indexes. We join all five tables in the above SQL query on the schema level as follows:

1. We join tables `dataset_work` and `work`, duplicating each work for each containing dataset, as illustrated in Figure 3.2. This transformation does not duplicate any rows because no works are contained in multiple datasets.

2. After renaming the `id` column of `work` to `work_id` and the `id` column of `expression` to `expr_id`, we join tables `work` and `expression` into `work_expr`. This is shown in Figure 3.3. Here we duplicate each work for each contained expression.

3. We rename `id` of `manifestation` to `mani_id` and `id` of `item` to `item_id`, then we join `manifestation` and `item` into `mani_item`, as shown in Figure 3.4. When importing Trove we only create one item for each manifestation, thus by joining we do not add a lot of redundancy (assuming there are no other very large datasets in the database for which this is the case).

4. We join `work_expr` and `mani_item` into `work_expr_mani_item`, see Figure 3.5. For Trove the relationship between manifestation and expression is one-to-one and if we assume other datasets are not too large we do not introduce a lot of redundancy.

All transformations have in common that one identifier column is removed and a primary key index is replaced by a B-tree index required for any referencing foreign keys (e.g. the `id` column of work to the `work_id` of `work_expr`). When joining tables on the schema level we increase the storage required, but this is only the case for (2). The change in storage required may actually be negative for the other three, since the most significant change for these is removing one identifier column. We do not know the net change, but the query in Listing 3.2 can now be executed within 10 minutes. We can also use the new structure for queries q_3 and q_4, even though those queries do not require that manifestations are joined with items (this is rather insignificant since manifestations of Trove books only have one item).

Consider query q_2. We simply define an index with key (`dataset_id`, `publ_year`). The RDBMS can now consider this index when deciding its execution plan, because the key can be used when scoping the results to a particular dataset (Trove), and the ordering by year defines an implicit grouping. In fact the query can be answered by counting pointers to rows in the B-tree, without reading any row. Maybe a temporary table in combination with a full table scan will still be performed. Since the number of groups is very low, the temporary table can fit in memory and even though we have an index we still have to consider almost every row in the database. We do not worry too much about this, since the RDBMS is sophisticated in deciding an execution plan (e.g. takes into account data-statistics). This query is now executed within one minute.

Optimizations around q_1 without filtering. To optimize q_1 we have to perform more joining on the schema level, embed the categorization into buckets in the database and forget about the `excluded_creator_ids` and `title_inclusion` filters. We first consider q_1 without any filters. We have to return the number of items for each location of publication (unveiling) for each bucket, within a particular dataset, as shown in Listing 3.3. The SQL expression `bucket(i.publ_year)` computes one of five buckets (0 through 4) based on the year at which the manifestation was unveiled. We uniformly map the range $[\min, \max]$ to $[0,4]$ where min and max are the minimum and maximum year at which a manifestation within the current

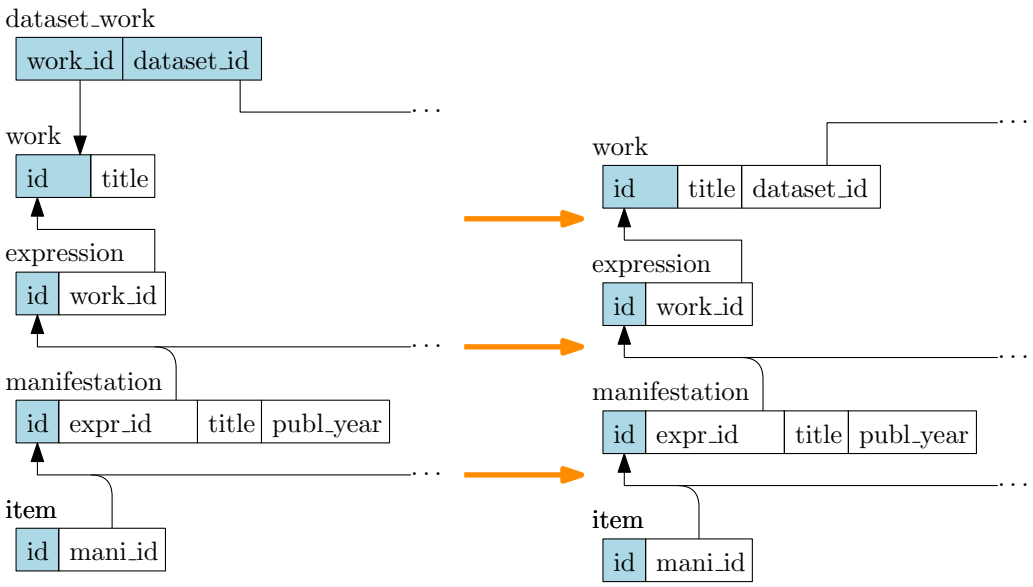


Figure 3.2: We join the tables `dataset_work` and `work` on the schema level.

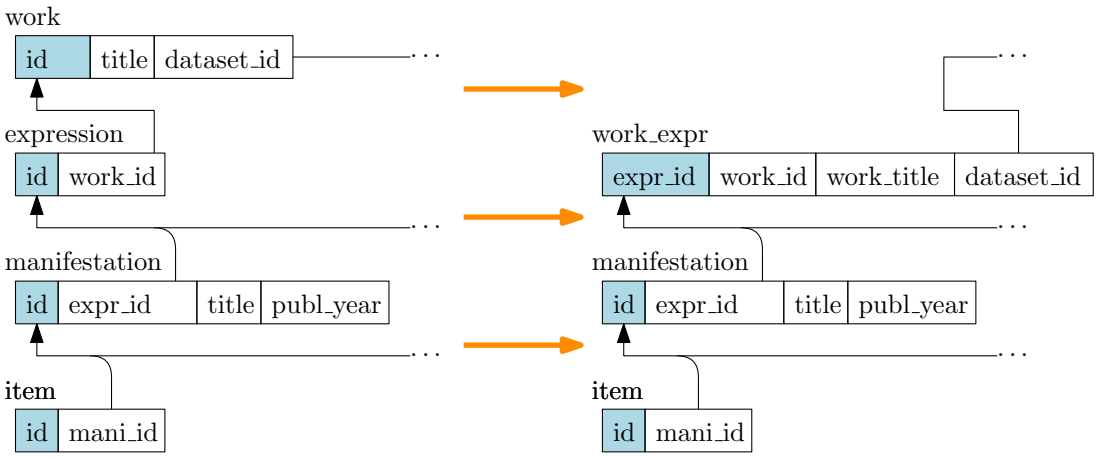


Figure 3.3: We join the tables `work` and `expression` on the schema level into table `work_expr`.

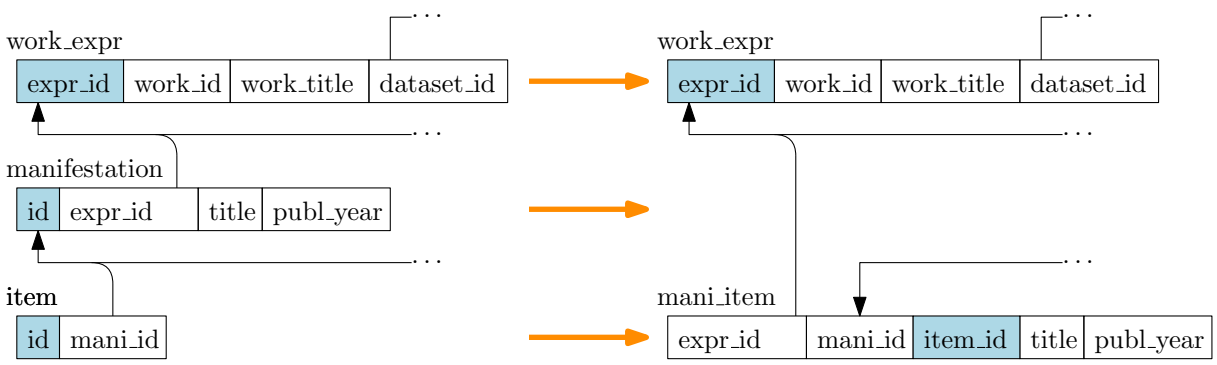


Figure 3.4: We join the tables `manifestation` and `item` on the schema level into table `mani_item`.

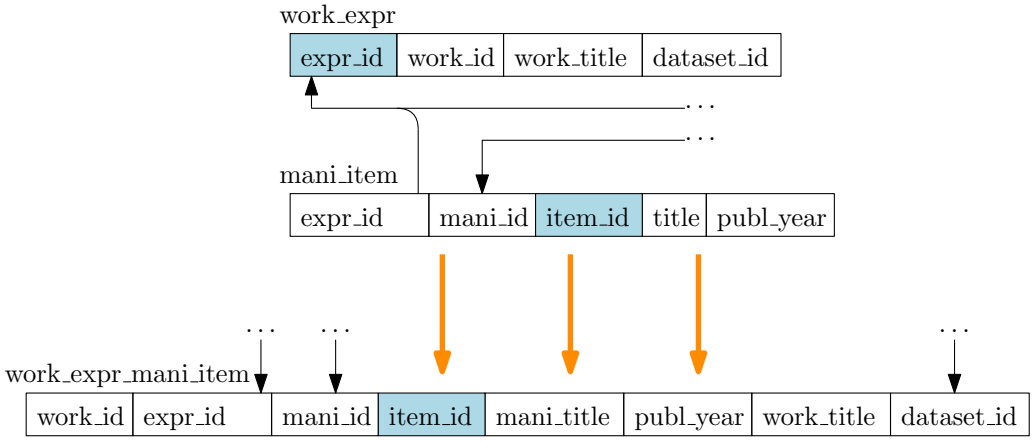


Figure 3.5: We join `work_expr` and `mani_item` on the schema level into `work_expr_mani_item`.

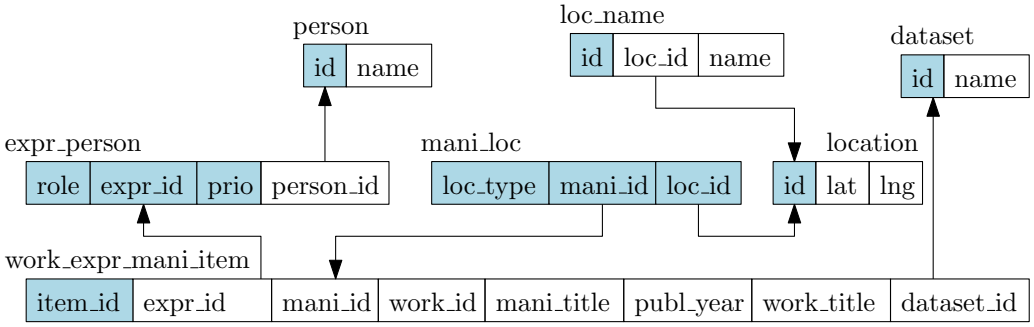


Figure 3.6: An overview of the database schema just before optimizations around q_1 are applied.

dataset was unveiled. We can retrieve min and max for any dataset in constant time, due to having defined the index with key (`dataset_id`, `publ_year`) earlier and MySQL optimizing queries for extremes to lookups if an applicable index is found (the index (`dataset_id`, `publ_year`) is applicable, but would not be if we did not scope querying for extremes to the current dataset).

```
SELECT mani_loc.loc_id, bucket(i.publ_year), COUNT(*)
FROM work_expr_mani_item i
  INNER JOIN mani_loc ON mani_loc.mani_id = i.mani_id
WHERE dataset_id = 5
GROUP BY mani_loc.loc_id, bucket(i.publ_year)
```

Listing 3.3: A SQL query to retrieve the leaf clusters for an unfiltered dataset with identifier 5.

This query does not finish within 60 minutes because there is no index that MySQL can use and the RDBMS has to resort to using a temporary table to store results and perform a full table scan of a joined table. The schema we have at this point is seen in Figure 3.6. The size of rows of `work_expr_mani_item` can still be reduced since some data types are unnecessarily large and the column `work_title` is never read from. Reducing the size of rows will result in full table scans being quicker because we can then fit more rows in less disk pages. We half the size of `dataset` identifiers, reducing the maximum number of datasets from 2^{31} to 2^{15}. Furthermore, we half the size of the data type of `publ_year`, reducing its range to $[-2^{15}, 2^{15} - 1]$. Then we move the column `work_title` to a different table, as shown in Figure 3.7. Even though we require an extra column of identifiers `work_id` in `work_info`, we now no longer duplicate the title of a work for each contained expression. We reduced the size of rows in `work_expr_mani_item`, but query q_1 still does not finish within 60 minutes.

We can still join the location of publication (unveiling) of manifestations (in `mani_loc`) with `work_expr_mani_item` on the schema level, add a column for the categorization into buckets, (pre-)compute these and define an appropriate index. These changes to the database schema are shown in Figure 3.8. We now have embedded buckets at the database level. This means that if we wish to change how items are categorized, we have to recompute these buckets. Alternatively, if we wish to support a different categorization at the same time, we can add another bucket column, pre-compute its content and define an additional index. At the moment there is no functionality

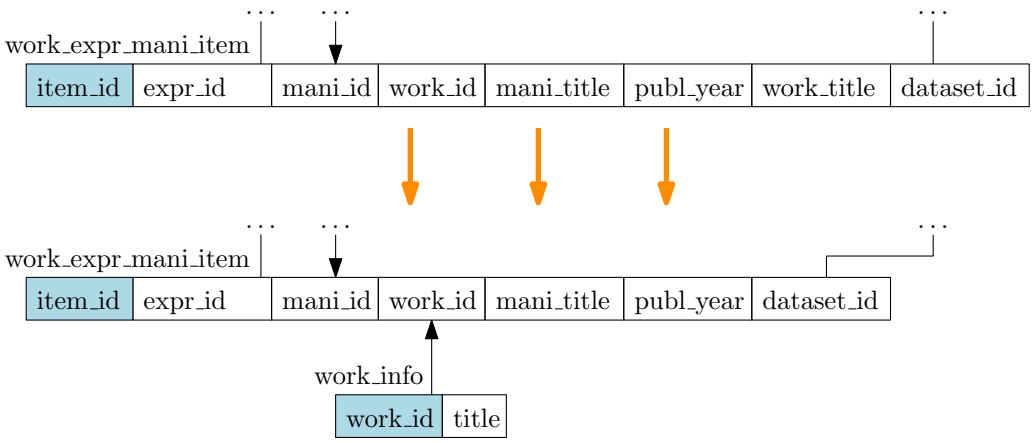


Figure 3.7: We move column `work_title` to a seperate table.

for choosing different categorizations in GlamMap, thus hard-coding this categorization deeper is fine for now. The key of the B-tree we create on `work_expr_mani_item` is (`dataset_id`, `publ_year`, `publ_loc_id`, `bucket`). With these changes, we can perform the query (without filters) to find the number of items for each location for each bucket in 4 minutes.

Tuning. The database of Trove is 31GB on disk (the table `work_expr_mani_items` and its indexes 8GB). The server has 32GB of memory available but MySQL only uses at most 2GB of memory. We used the linux command 'du -h' in the data directory (`/var/lib/mysql`) on the server. The setting `innodb_buffer_pool_size` was changed from its default of $1/8 \cdot 1024^3$ to $20 \cdot 1024^3 - 1$ bytes. This way important tables and indices can fit in memory. We furthermore change `innodb_write_io_threads` and `innodb_read_io_threads` from 4 to 64. This determines the maximum number of threads used by the InnoDB engine to read and write data from and to disk, respectively. InnoDB by default automatically chooses the actual number of threads (up to the specified maximums) and we trust InnoDB will saturate the disk now. Before these optimizations the query q_1 without filters on the Trove dataset did not terminate within 5 minutes. Note that the Trove dataset is defined to be the books in Trove published in the 8000 most frequently occurring locations (this is 95% of Trove). After these optimizations we obtain the running time shown in Figure 3.9, respectively. This graph depicts the time in milliseconds between requesting a clustering of Trove (query q_1) from within GlamMap and having all glyphs rendered (the *response-time*) and is averaged over five samples. The filter in the test after tuning sets `publ_year_max` to 1938, splitting the Trove dataset approximately in two halves. In Figure 3.9 the average response-time for filter requests is lower than for non-filter, which is plausible since less items have to be counted. With tuning we still do not achieve interactive times.

More optimizations? It is safe to assume (by tuning and its effect) that the relevant part of the database (the `work_expr_mani_items` table and its indices) is now kept in RAM by InnoDB. We can conclude we have now arrived at a CPU bottleneck for query q_1 (with no or year filters). Note that we can still partition this table and we do not apply parallel query execution, but we leave this for future work. We still have to optimize for filtering by creator.

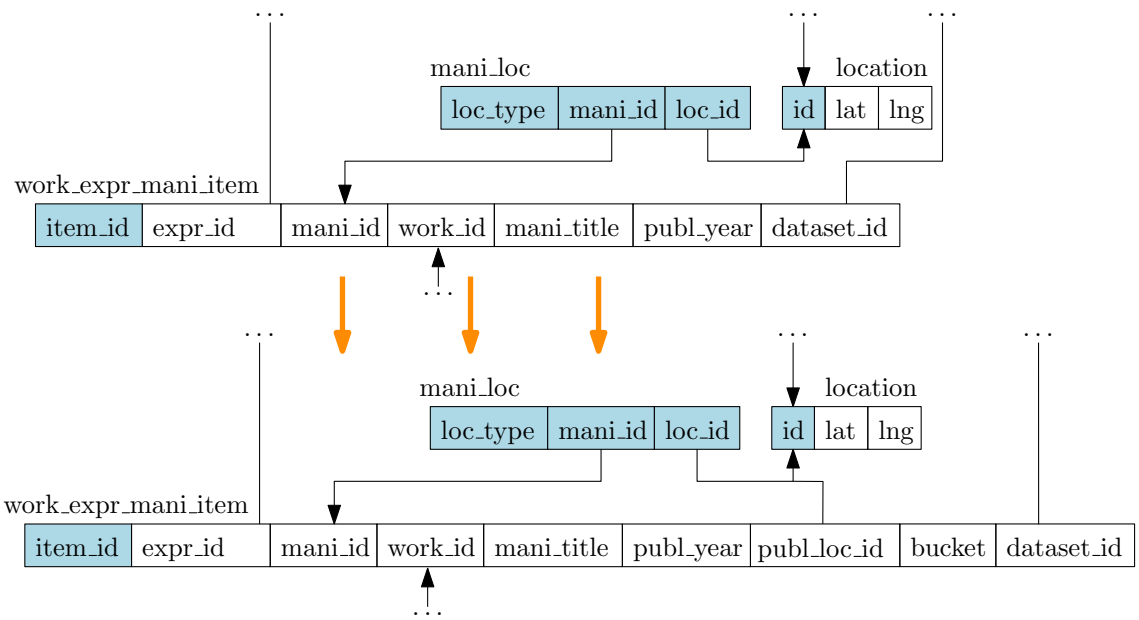


Figure 3.8: We move the location of unveiling (publication) of manifestations from `mani_loc` to `work_expr_mani_item` and add a column `bucket`.

Query q_2 does not have to be optimized any further because it is unaffected by filters and its answer can be cached (storing for subsequent use). When building the cache (executing q_2 once) we can use the index for q_1 to count pointers grouped by year of unveiling. This takes less than two minutes.

Queries q_3 and q_4 still need to be optimized. This is discussed in Chapter 4.

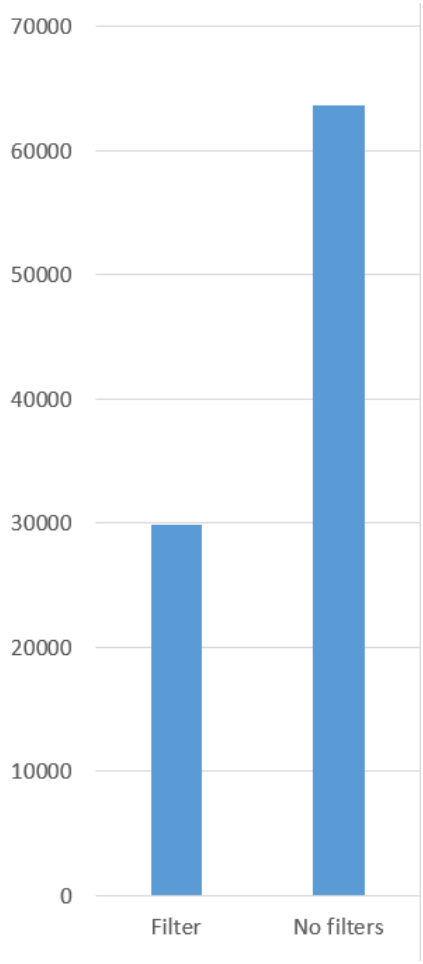


Figure 3.9: The time in milliseconds between requesting an overview of Trove and having a full rendering.

3.4 Optimizing the clustering algorithm

Computing the clustering of the full Trove dataset with one-hundred thousand locations is slow and can be improved. Even when only 8000 out of 100000 initial clusters with 59 and 60 million items, respectively, are passed to the clustering algorithm it does not terminate within 10 minutes. We measure the time spent in various parts of the algorithm:

1. Initialization of the NNG.
2. Computing the initial $O(n)$ pairs of clusters that may have to be merged when zooming out.
3. Computing the size of glyphs.
4. Merging clusters, updating the NNG and pairs of clusters that will have to be merged when zooming out.

More than 80% of the time was spent computing the size of glyphs, so we focus on making this sub-algorithm faster. We first implemented two improvements, reducing the number of layouts considered and the time spent computing them. Since this did not prove sufficient, we need a third improvement, or rather a replacement of this sub-algorithm.

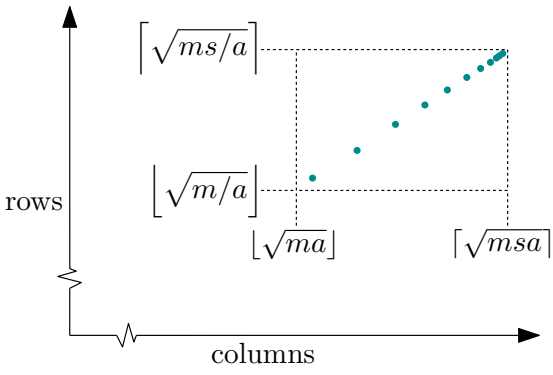


Figure 3.10: The arrangements considered are restricted from the rectangle to dots along its diagonal.

Reducing the number of layouts considered. Figure 3.10 illustrates the smaller set of layouts we consider. Each layout considered is represented by a point in the plane. Instead of considering a rectangle of layouts, the idea is to only consider non-uniformly spaced layouts on the diagonal. We modify the algorithm for computing glyph interior sizes, as specified in Algorithm 1. The algorithm for computing one layout (COMPUTELAYOUT) is the same as before, but the number of layouts considered by the containing algorithm (COMPUTEGLYPHINTERIORLAYOUT1) has reduced. The layout last considered and computed (line 6) is returned. We start with the layout represented by the bottom-left green point in Figure 3.10. As long as A is too small (cannot accommodate for m items) we recompute it with a slightly larger number of rows and columns. Successive layouts must also account for the "cut-off" in rounded corners of previous layouts, in terms of number of items they must be able to represent. If r_2 is too small, `no_rows` and `no_cols` may not actually increase. We remedy this by incrementing in this case the one which best preserves the aspect ratio a of `no_rows` and `no_cols`.

This approach converges to the smallest layout that can accommodate m items and we never consider too large layouts. This method may produce less "nice" layouts than the original. In particular, the original algorithm considers layouts where the aspect ratio of rectangles representing items deviates more from a than this method (further away from the diagonal). This

Algorithm 1 Computing the size of glyph interiors

Precondition: $r_2 \in [0, 1/2]$, $0 \leq m$ and $0 < a$
Postcondition: the layout A can accommodate for at least m items

1: **function** COMPUTEGLYPHINTERIORLAYOUT1(m, r_2, a)
   m is the number of items represented by the glyph
   r_2 is a parameter for the roundness of the glyph interior (0 is a square, $1/2$ is a circle)
   a is the ideal aspect ratio of rectangles (each representing one item) to lay out within the glyph interior (the actual aspect ratio may differ since a layout must be a square)
2: $m' \leftarrow m$
3: **repeat**
4: no_rows $\leftarrow \left\lfloor \sqrt{m'/a} \right\rfloor$
5: no_cols $\leftarrow \left\lfloor \sqrt{m'a} \right\rfloor$
6: $A \leftarrow$ COMPUTELAYOUT(no_rows, no_cols, r_2, a)
7: $m'' \leftarrow$ the number of items that A can accommodate for
8: $m' \leftarrow m' + m - m''$
9: **until** $m \leq m''$
10: **return** A

deviation may still be too small to notice while the first row of such layouts is fuller than the first row of layouts of the improved version. This "niceness" are not important because the first two improvements still did not enable the clustering algorithm to terminate within 10 minutes (even on only 8000 out of 100000 locations with 59 out of 60 million items) and thus this improved version is not sufficient.

Reducing the time spent computing layouts. The algorithm that computes the size of one layout does not consider vertical symmetry. In Figure 3.11 we can see symmetry between groups (of rows) 1 and 3. If we have computed the number of columns for rows in group 1 then the reverse of this is the number of columns in group 3, and vice versa. The original algorithm uses horizontal symmetry within rows, but does not use this vertical symmetry. The roundedness of glyph interiors for which we compute layouts is determines by the parameter $r_2 = 14/60$, therefore the number of rows in group 1 is approximately $14/60$ of all rows. We change the representation of a layout to only the number of columns for rows in group 1. We can derive the number of columns in groups 2 and 3 if we additionally store the number of rows and columns of the (grid of the) layout. This change in running time and representation size cannot be asymptotically seen, but improves them by approximately a factor of $14/60$. This improvement was

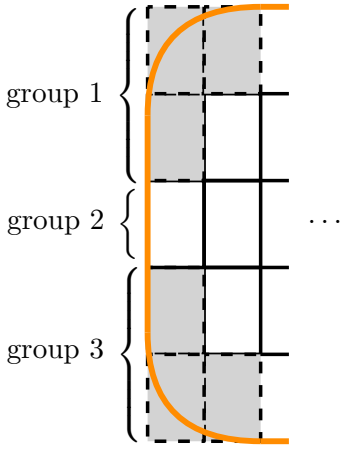


Figure 3.11: Computing rows in group 1 and 3 is symmetric. For rows in group 2 no "cut-off" has to be computed.

not enough because the clustering algorithm still takes at least 10 minutes to terminate (even on only 8000 locations with a total of 59 million items) and we require interactive speeds.

Figure 3.12: A glyph representing 18 items.

Replacing the glyph size computation algorithm. In this approach we pretend glyphs are squares when laying them out ($r_2 = 0$) and to simplify things even further we hard-code $a = 1$ (the aspect ratio of rectangles to lay out, each representing an item). This results in the simple $O(1)$ time algorithm shown in Algorithm 2. The representation of a layout now requires $O(1)$ storage. Even though the layout is computed for rectangles, we still want the visual fidelity of having rounded corners. For this reason we simply clip the square in which items are laid out to a rounded one, as shown in Figure 3.12. Since items are rather large, we can see that items are cut by corners. If items were to be smaller, they could be completely hidden under rounded corners. In this respect the surface of glyphs is not completely accurate any more, but this holds for all glyphs and thus their size relative to others is preserved. If we simply scale all glyphs this surface inaccuracy is completely remedied. It could be that a large portion of squares with the same colour (representing items with the same category) are hidden behind a corner and thus estimating the size of categories relative to others within a glyph may be harder. This problem is not very significant however, because relatively very little rectangles can completely disappear behind rounded corners. Now the clustering algorithm finishes in a few seconds on 8000 locations with 59 million items.

The clustering algorithm does not terminate within 10 minutes on the input with all locations (60 million items over 100000 locations), but note that 95% of all items in Trove are contained in 8000 locations. We can compute the clustering for approximately 95% of all items in a few seconds. We know that the geocoding contains errors, however. It could be the case that an error-free geocoding only has several thousands of locations, in which case no optimization is required. Otherwise, we do require to optimize the clustering algorithm further. For now we assume that a correct geocoding produces 8000 or less locations and therefore this optimizations is sufficient for interactive speeds.

Algorithm 2 Computing the size of glyph interiors

Precondition: $0 \leq m$
Postcondition: the layout A can accommodate for at least m items

1: **function** COMPUTEGLYPHINTERIORLAYOUT2(m)
   m is the number of items represented by the glyph
2: $r \leftarrow \lfloor \sqrt{m} \rfloor$
3: $c \leftarrow r$
4: **if** $rc < m$ **then**
5: $c \leftarrow c + 1$
6: **if** $rc < m$ **then**
7: $r \leftarrow r + 1$
8: **return** a layout A with r rows and c columns

3.5 Optimizing rendering glyphs

The image-based rendering technique is slow and makes the web-page unresponsive because:

P1: Several unnecessary conversions of image data are performed.

P2: Images are not reused as much as possible (redrawn too much).

P3: Images are drawn greedily.

P4: Each item represented by a glyph is a distinguishable square in the glyph.

Problems *P1* and *P4* make rendering individual glyphs slow and *P2* and *P3* make the number of glyphs rendered high. We solve this by:

S1: Only drawing images the first time they can be seen.

S2: Applying incremental drawing (yielding to user input while rendering).

S3: Reusing images across different zoom levels.

S4: Grouping squares (that represent items in the same category) together.

Solution *S1* and *S3* tackle problems *P2* and *P3*. Solution *S4* tackles *P4* and solution *S2* ensures responsiveness of the page while rendering.

Changes and results. When the visualization is initially invalidated, we do not draw all three images (each with a different border colour) for all glyphs in the hierarchical clustering, but rather draw only one image for each glyph that can be *seen*. A glyph can be seen if its axis-aligned bounding box intersects the viewport and that glyph is visible at the current zoom level or it is being previewed. Restricting which images are drawn also applies to any invalidations after the page is loaded. Determining which glyphs lie in the viewport is not done using an intelligent data structure, instead we simply check for each glyph that should not have been split at the current zoom level whether it intersects with the viewport. Whenever an image is drawn, we store it

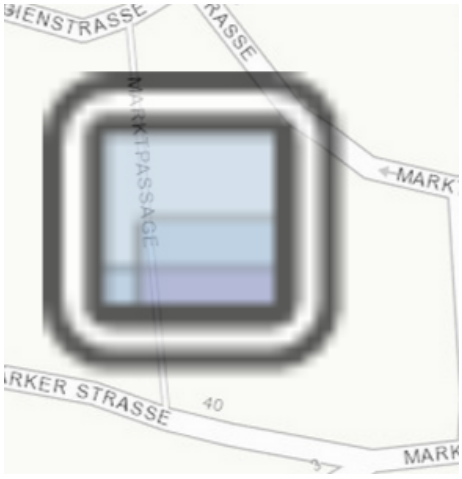
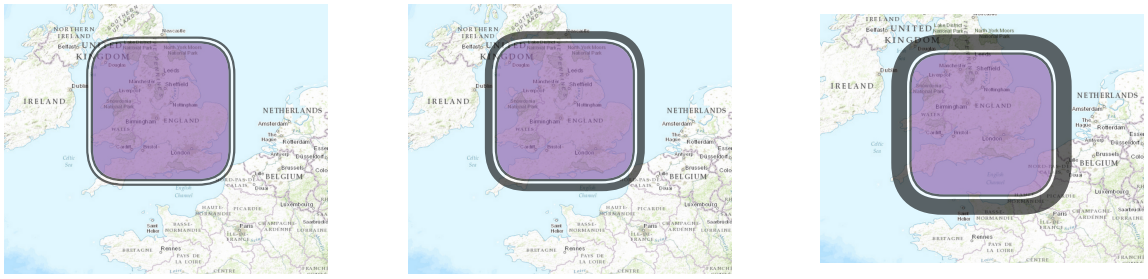


Figure 3.13: Images may become blurry when zooming in.

and reuse it whenever the respective glyph can be seen again. We yield to user input every 10 milliseconds while rendering, causing the web-page to be responsive while rendering. We do not actually remove unnecessary conversions of image data, but by reusing images, not drawing the entire hierarchical clustering immediately and grouping squares with the same colour we already gain sufficient performance. The disadvantage of this method is that glyphs can become blurry when zooming in far. If a glyph was rendered at a high zoom level and we zoom in on this glyph we can see this effect (Figure 3.13). We no longer draw three images of each glyph, but always draw one, even though the mouse pointer is over a glyph or we click one.

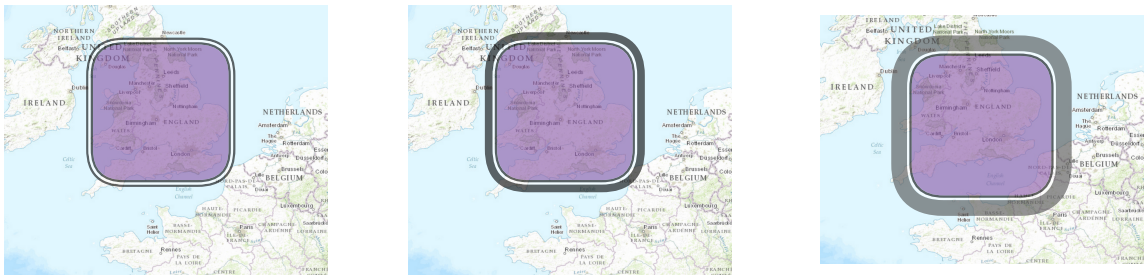
3.6 Glyph compression

When scaling the area of glyphs linearly in the number of represented items, additionally the amount of items per location differs by a factor in the thousands and we want small glyphs to be easy to spot then a large glyph can cover up an entire continent or half of the viewport. More generally, the map does not have the right "fullness" in this case. It is hard to perceive the location of glyphs that are very large and users may get disoriented. We *compress* glyphs that are too large, by downscaling their interior, to solve this problem. Since we want the map to show the geographic distribution of items, if we do not provide a clear visual indication of this compression then users may perceive the density of items at a compressed glyph incorrectly. We use a simple model with three levels of compression (0, 1 and 2). Level 0 indicates that no compression has been applied. Levels 1 and 2 indicate a small and large, respectively, compression. Using three levels is sufficient to obtain the right map "fullness" for all viewport configurations when showing Trove. The compression level of a glyph is encoded by its border pattern and colour and we tested three different designs of this encoding. Compression level zero, one and two scale the area of the glyph interior by 1, $25/39$ and $4/9$, respectively. The compression level is defined by thresholds on a per-dataset basis, as listed in Table 3.1 (for a glyph representing m items we choose the highest compression level with threshold t such that $t \leq m$).



(a) Level 0 (b) Level 1 (c) Level 2

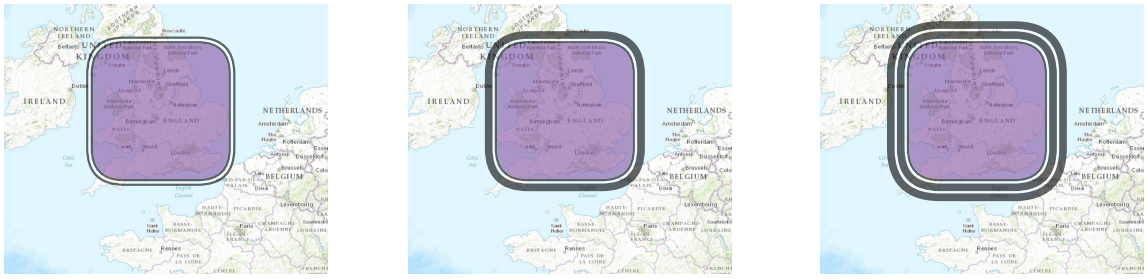
Figure 3.14: Compression level encoded in glyph border (design 1), discriminating between levels only on outer border thickness.



(a) Level 0 (b) Level 1 (c) Level 2

Figure 3.15: Compression level encoded in glyph border (design 2), discriminating between levels on outer border thickness and colour.

Designs. The first, second and third design are shown in Figure 3.14, 3.15 and 3.16, respectively. Level one increases the size of the outer black border. In design 1 and 2 the outer border becomes thicker at compression level 2. Distinguishing between levels 1 and 2 solely on thickness (in design 1) made glyphs with compression level 1 and 2 look too much alike when showing a lot of glyphs on the map at the same time. In design 2 we attempt to further discriminate between levels 1 and 2 by changing the outer border colour from black to a lighter gray shade. This did not solve the problem of design 1 and we need a clearer discriminator between levels 1 and 2. In design 3 the border for compression level 2 is simply the pattern for level 1 repeated once. Effectively we add a white border to compression level 2 of designs 1. Design 3 is the best because it provides the best discrimination between compression levels, even for datasets and regions with a high density in terms of the number of visible glyphs.



(a) Level 0 (b) Level 1 (c) Level 2

Figure 3.16: Compression level encoded in glyph border (design 3), discriminating between levels on outer border pattern.

	Risse	Trove
Level 0	0	0
Level 1	$2 \cdot 10^3$	10^6
Level 2	$4 \cdot 10^3$	10^7

Table 3.1: A table showing the glyph compression thresholds per dataset.

3.7 Changes to glyph borders

The border design used by the GlamMap prototype can make glyphs hard to distinguish from the underlying map, as shown in Figure 3.18. This is especially the case for the top border. The light rectangles and the adjacent inner white border have a colour intensity very similar to the map. The outer black border is too thin and the glyph is best distinguished from the underlying map by the purple area and black interior. The border thickness in the GlamMap prototype also changes slightly when the zoom level changes. We improve the border design altogether by increasing the border-thickness, making it completely independent from the zoom level and adding an inner black thickness. Making the border thicker is easy, but reflecting the changes in the clustering algorithm is slightly more cumbersome.

Propagating the new border design to the clustering algorithm. We change the formula for the size of a cluster x (with interior radius r_x and border thickness b_x) from $2^{-i}1.25^i(b_x + r_x)$ to $2^{-i}(b_x + 1.25^i(r_x))$. The zoom level i at which two clusters x and y begin to intersect (touch) is

$$f(i) = c - 2^{-i}(b_x + b_y + 1.25^i(r_x + r_y)) = 0 \tag{3.1}$$

where

1. b_x and b_y are the border thicknesses of cluster x and y, respectively (taking into account border size changes due to compression);
2. r_x and r_y are the radii of the interiors of the glyphs of x and y, respectively;
3. c is the L^∞ distance between cluster x and y.

(a) Old

(b) New

Figure 3.18: A comparison of the border design used in the GlamMap prototype and the one used now.

We want to solve Equation 3.1 for i, but there no longer is an algebraic solution. Luckily it can be solved using Newton's method. We need only a constant number of digits of precision due to floating point representation thus this takes $O(1)$ time. We do not approximate all digits in the floating point representation, but iterate until the solution changes by less than 10^{-5}. This gives a slight improvement in terms of speed at the cost of accuracy, but is easy to implement. The initial guess for i is $\log_{5/8}(c/(b_x+b_y+r_x+r_y))$, which is the exact solution of the equation expressing the zoom level at which two clusters begin intersecting in the GlamMap prototype (Equation 2.1). We do not have convergence issues, but the use of floating point numbers does lead to a problems. Function $f(i)$ is non-decreasing and approaches a horizontal line. As i increases, the derivative of $f'(i)$ gets closer to zero. When dividing a number by a very small one (in particular $f'(i)$ for high i), the mathematical result may not be representable by floating point numbers (too large). In this case a special (non-finite) value representing ∞ is produced. This error is silent and does not cause a crash, but rather leads to this application of Newton's method iterating forever. We solve this by checking explicitly for the ∞ value and if this is the case we reset the approximation for i to 10. This number is approximately the average zoom level at which clusters merge. We have not encountered any instance yet on which this method fails or requires more than 10000 iterations, thereby experimentally verifying it.

3.8 Changes to side panels

The author panel has been adapted to more clearly show the relative amount of books published per author. The old and new author panel are shown in Figure 3.19 and Figure 3.20, respectively. The details panel has been adapted to use paging. The timeline automatically computes labels, to be able to deal with arbitrary datasets. In particular datasets for which no manual labels have been defined are now supported. Due to the new design of the year panel the specific tuning of labels for Risse was lost, as can be seen in Figures 3.21 and Figures 3.22. Paging still should be implemented in the author panel, since the millions of authors in Trove cannot be displayed at once. While upgrading GlamMap title filtering was not fully ported and this field does not work. Items in the details-on-demand panel now contain a link to their containing work, on Trove's

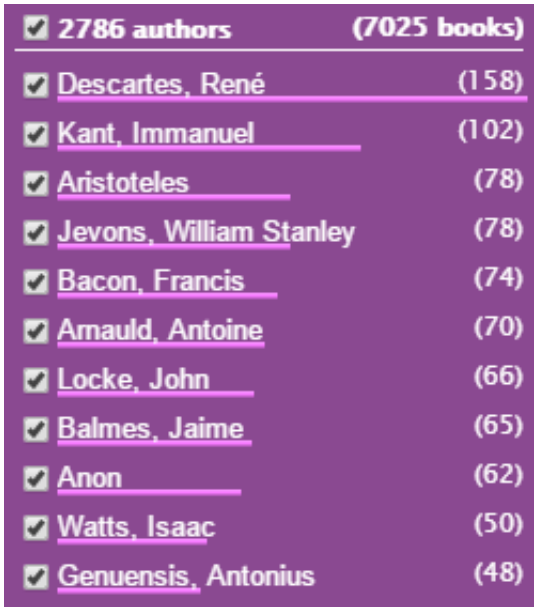


Figure 3.19: The old author panel.

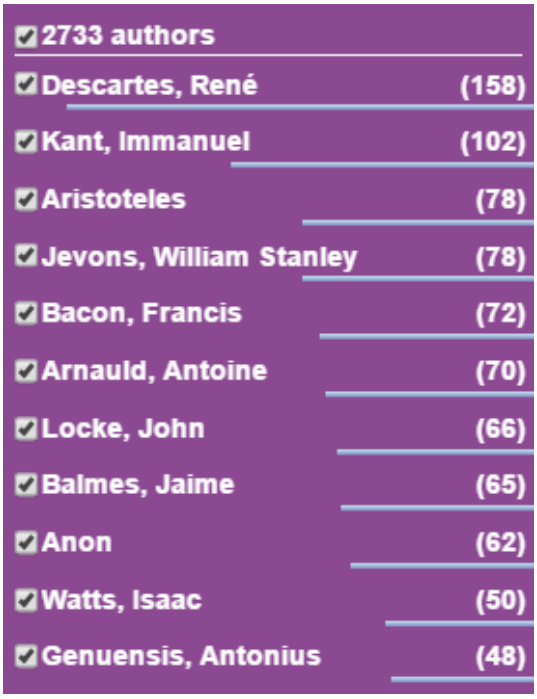


Figure 3.20: The new author panel.

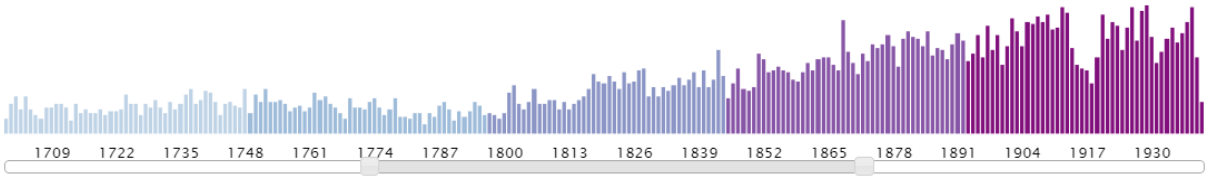


Figure 3.21: The old timeline panel.

official website. Note that although this is not reflected in the database schema discussed here, we do in fact associate works with their respective Trove work identifier.

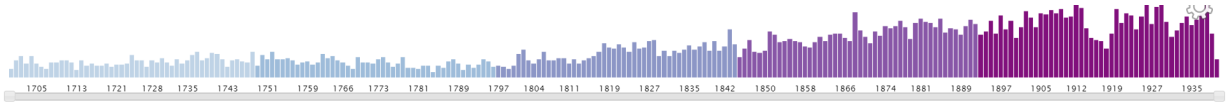


Figure 3.22: A new timeline panel.

Chapter 4

Results and discussion

An overview of Trove can be seen in GlamMap [2], rendering is fast, enabling smooth zooming and panning, but filters cannot be applied at interactive speeds. We categorize other results in four categories: data-quality, data-access, clustering, rendering and visualization. The second category is a big sticking point when integrating Trove with GlamMap.

4.1 Data quality

Trove allows books to have multiple authors and books to be published at one or more locations, but this is not reflected in GlamMap.

Not all locations can be automatically geocoded and it is unknown how many are correctly geocoded. It seems that the majority is correct since big quantities of books were published in Europe and the United States of America, as can be seen in Figure 4.3.

The disambiguation scheme used when geocoding leads to errors and should be improved. Consider the example "London", which may refer to a place in Canada, but is better known as the capitol of the United Kingdoms. A book aged from around the 16th and 17th century reporting on colonies is likely to have originated from somewhere in Europe and hence London will probably refer to the place in the United Kingdoms and not to a place in Canada. Resolving such an ambiguities requires that book text semantics are taking into account and is difficult to do fully automatically. Crowd-sourcing will allow more accurate location data to be collected, even though Trove is frequently updated and improved itself.

By associating bibliographic records in GlamMap with other databases, we allow users to easily view related information. For example: Trove already links its books to the WorldCat dataset. This enables us to create hyperlinks in GlamMap linking to WorldCat for the Trove dataset.

We do not detect duplicate authors with different names. This can be done automatically by finding two authors for which the last names are equal, their initials and first names are compatible and either they have exactly the same age or they published books around the same time period. A crowd-sourcing approach would also be appropriate here.

The year at which books are published is sometimes incorrectly imported. In particular some

books are erroneously thought to be published in the years 0-100 while most of them actually were published somewhere after 1700. This causes the histogram in the year panel to show a gap between the year 100 and 1700. For some books the exact year of publishing is unknown, but the century is. These books are then interpreted to be published at the start of the century. We can see this clearly for the years 1800 and 1900 in Figure 4.1. For some of these books a range of years (during which books were approximately published or issued) is provided by the Trove-service. We can take the average of this year, to get rid of many of these artefacts. This figure shows the 95% of the books in Trove published at the 8000 most frequently occurring locations of publishing.

4.2 Data-access

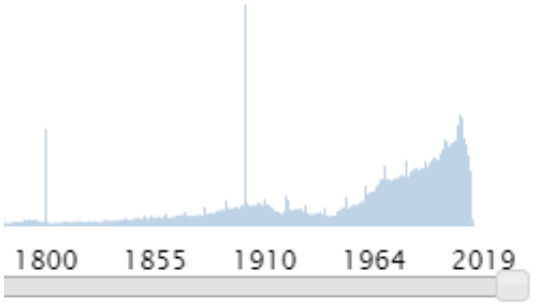


Figure 4.1: The histogram shown in the year panel with the biggest bulk of books.

We know the relevant part of the database (the `work_expr_mani_item` table and indices of 8GB) fits in memory, and showing overviews for q_1 with no filters or year filters can be done in 10 to 60 seconds. By the effect of server tuning it is safe to assume InnoDB actually places the relevant part of the database in RAM. To this extent we have established a CPU bottleneck for the clustering query with no or year filters. Furthermore, combing the `title_inclusion` filter with others seems difficult. We still have to optimize the filters `title_inclusion` and `excluded_creator_ids`. This storage architecture is well-suited for GlamMap because metadata can be stored naturally in an organization of interrelated tables and more importantly it provides the necessary data structures and algorithms to access data quickly.

Further B-tree based optimizations. We have not focused on optimizing the queries for the author and details-on-demand panel, q_3 and q_4, respectively. Query q_3 can be optimized by applying the technique of joining authors (in `expr_person`) with `work_expr_mani_item` on the schema level. This introduces one column named `creator_id` in `work_expr_mani_item`, identifying the creator (author) of expressions. Then we can add an index with key (`dataset_id`, `publ_year`, `creator_id`) to optimize q_3 (except on the filter `title_inclusion`). We can add `creator_id` to the keys of indices for other queries to optimize for the `excluded_creator_ids` filter. The filters `publ_year_min`, `publ_year_max` and `publ_loc_ids` can be optimized by adding the columns `publ_year` and `publ_loc_id` to index keys. For q_4 we want to add `publ_loc_id` to index keys for the appropriate queries. Note keys of indices may share prefixes. For example the key of the index for q_1 (after adding `creator_id` and `publ_year`), could be (`dataset_id`, `publ_year`, `creator_id`, `publ_loc_id`, `bucket`) while we already have an index with key (`dataset_id`, `publ_year`) for q_2. In this example the index for q_1 can in fact be used for all four queries. This is not necessarily desired, since having larger index keys will increase the depth of the B-tree and if we only need a prefix of the

index key we need to traverse more levels to iterate over all row pointers. In the future we could test whether sharing the exemplified large index for q_1 (for less storage size, but a larger time required to traverse it) is "better" than using multiple small indices (for less redundant parts, but a smaller time to required to traverse each).

Implementing the title filter (search). Optimizing the filter `title_inclusion` cannot be done by adding the `mani_title` column to index keys. A B-tree first orders rows on the most significant (first) key column. Within this column we first order on most significant digits first (for integer columns). Suppose we want to select only manifestations including the text 'mozart' in their title, there is a B-tree index with key (`mani_title`) and we attempt to use this index to answer the query. The subsequence we are looking for may be at the least significant position of titles, therefore to be sure we find all such rows we have to traverse all branches of the B-tree index and we may be better off performing a full table scan. MySQL supports full text indices, which are better suited for optimizing this filter. However, such an index cannot produce an ordering of rows we desire for queries q_1 and q_2, and we cannot quickly answer queries with title filters that produce a large set of rows. Fortunately for q_1 and q_2 produce a fairly small amount of groups (25000 and 2500, respectively, for 95% of the books in Trove) and we only have to count the number of rows per group. If we count this with a table with one integer per group, we require little auxiliary storage.

Combining the title filter with other filters. We do not know any way of efficiently supporting data-queries with a `title_inclusion` filter in combination with another. We can redefine `title_inclusion` to select manifestations with titles starting with a particular sequence of characters. This way we can use B-tree indices and combine this filter with other filters, however this does not solve the problem of combining full text-based searching with other filters. The root of this problem is the fact that a full text index cannot be combined with a B-tree index. More precisely, MySQL cannot create a full text index for all possible sets of rows produced by a B-tree index and vice versa. We can of course intersect rows selected by the `title_inclusion` filter (say list L_1) with rows selected by other filters (say list L_2). List L_1 is not sorted so we cannot simply compute the intersection in $O(|L_1| + |L_2|)$ time without additional non-constant storage, but we can using a temporary hash-table. We then put all rows in one list in the hash table and report rows in the other list that are not in the hash table. The size of this temporary table needs to be big enough for millions of rows. This should not be a problem since the server has 32GB of RAM and the relevant part of the database (the `work_expr_mani_items` table and its indices) is 8GB, leaving sufficient memory for temporary tables.

Parallelism to overcome CPU-bounds. We can still use the CPU better (in particular parallel computing). By using MySQL's partitioning we can reduce the size of search-indices and when searching such that partitions can be pruned we will gain query execution speed compared to not having partitions. This also enables us perform parallel execution of query q: for each partition p we execute q scoped to p, in parallel. Note that MySQL does not support parallel execution of one query, but can execute multiple queries in parallel (as is the case now). The disadvantage of this method is that if a partition is pruned we lose parallelism and furthermore we have to aggregate results of each partition ourselves. In the worst-case all partitions except one are pruned and we do not gain anything from this method of parallel query execution (PQE). Some

RDBMS systems have true support for PQE (true in the sense that they support it in their engine rather than leaving it to the application). They can use it to traverse different branches of a B-tree or scan a table in parallel, supporting PQE on a lower level with greater granularity. Example of RDBMS systems more advanced in this regards are Microsoft's SQL Server and Oracle. If our queries are still too slow we will need a distributed architecture (use multiple servers that work together).

Other storage engines. MySQL supports different storage engines besides InnoDB. We consider MEMORY and MyISAM (these ship with MySQL). The first stores tables and indexes in RAM and the application (GlamMap) has to ensure data is persisted and initialized. We have tuned the InnoDB storage engine to enable it to load the relevant part of the database (the `work_expr_mani_items` table and indices) into memory, thus the MEMORY engine requires at least that we initialize data, but likely does not gain us anything. The MyISAM was the default storage engine before MySQL 5.6, but was replaced by InnoDB. There are no significant performance differences between InnoDB and MyISAM. We have not considered third party storage engines.

4.3 Clustering

By improving the way the size of clusters is computed we have made the clustering algorithm sufficiently fast for 8000 locations (with 95% of the books in Trove). We do not know whether locations are geocoded correctly, but it seems like the number of distinct locations of publishing of books in Trove does not differ from 8000 by an order of magnitude.

If the number of locations after corrected geocodings is large (15000 or more) then optimizations of the clustering algorithm are still required. This would very likely imply that we need to improve the algorithm on a design level. For n locations we need to merge $n-1$ pairs of clusters. When a merge occurs the NNG and list of merge candidates needs to be updated. This update takes $O(n)$ time and this is the bottleneck in terms of asymptotic running time.

4.4 Rendering

Rendering is fast enough for Trove, but glyphs become un sharp at low zoom levels and rendering can still be done quicker. We claim that the image-based rendering technique is infeasible from a performance point of view and propose a vector-based rendering technique, solving the blurriness problem. In the GlamMap prototype the first step of drawing an image is rendering it to an off-screen buffer. At this point the browser already has an image in memory, but we subsequently convert it to an image format, base-64 encode it in a text string and let the browser load and render it. The performance of this method is bad due to the unnecessary conversions. In the GlamMap prototype three separate images are generated and stored for each glyph, each having a different border colour. Memory is wasted by storing each variation of one glyph in an image. Instead, we should represent drawings of glyphs using vector graphics (e.g. Scalable Vector Graphics). In this case the representation consists only of the individual vector graphics components (e.g.

path shapes for borders and the glyph interior). When the mouse pointer enters a glyph we can simply assign a property corresponding the colour of the inner white border, rather than having a separate image of the glyph ready. Using a vector-based rendering may very well be faster than the image-based technique (at least in terms of page loading time) due to it not performing unnecessary conversions. Note that the image based rendering technique draws the image to the off-screen buffer using vector graphics drawing commands.

4.5 Visualization

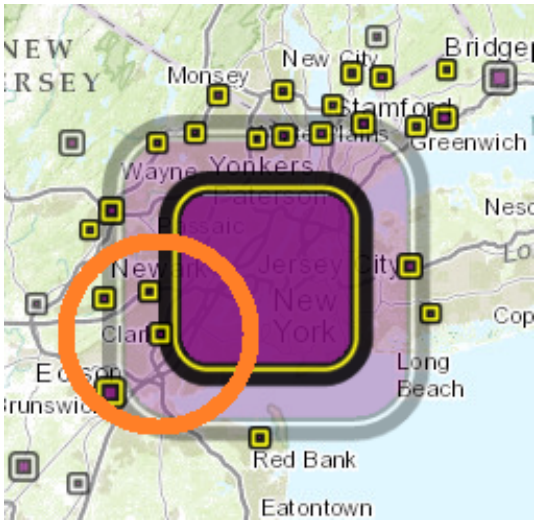


Figure 4.2: Clusters shown in the preview may overlap.

Figure 4.3 shows an overview of all books in the Trove dataset (only the 8000 locations at which most books were published, approximately 95% of the books in Trove). Copies of individual books in Trove and which library holds them can still be imported, opening another visualization opportunity. Panning and zooming over this data (filtering by geographical region) is supported at interactive speeds. Other filters such as selecting only books published within a specific year range and selecting specific authors are not interactive and not supported for the Trove dataset, respectively. The updated glyph design, specifically compression, allow for a map-based visualization with the right "fullness" at all zoom levels. Arbitrary datasets can be made available online through GlamMap due to the new architecture, but no options for tuning the visualization to these datasets are available to the user. The colour encoding is not as discriminating for Trove as it is for Risse, because books of Trove are not as uniformly spread over the range between their minimum and maximum year of unveiling. Ideally one can choose how to categorize data, but to achieve a feasible performance this has to be hard-coded into the database. Instead we can define a predefined set of categorisations from which the user can choose. The other smaller problems with the visuals are:

1. When showing the preview of a cluster at zoom level i overlap occurs (as shown in Figure 4.2), but this is no longer the case at zoom level $i+1$.
2. There are still some bugs with respect to sizing of glyphs in the user-interface, but they are hardly noticeable.
3. We use two different map projections in the clustering algorithm and when rendering glyphs. This should lead to inconsistencies, but these are again hardly noticeable.

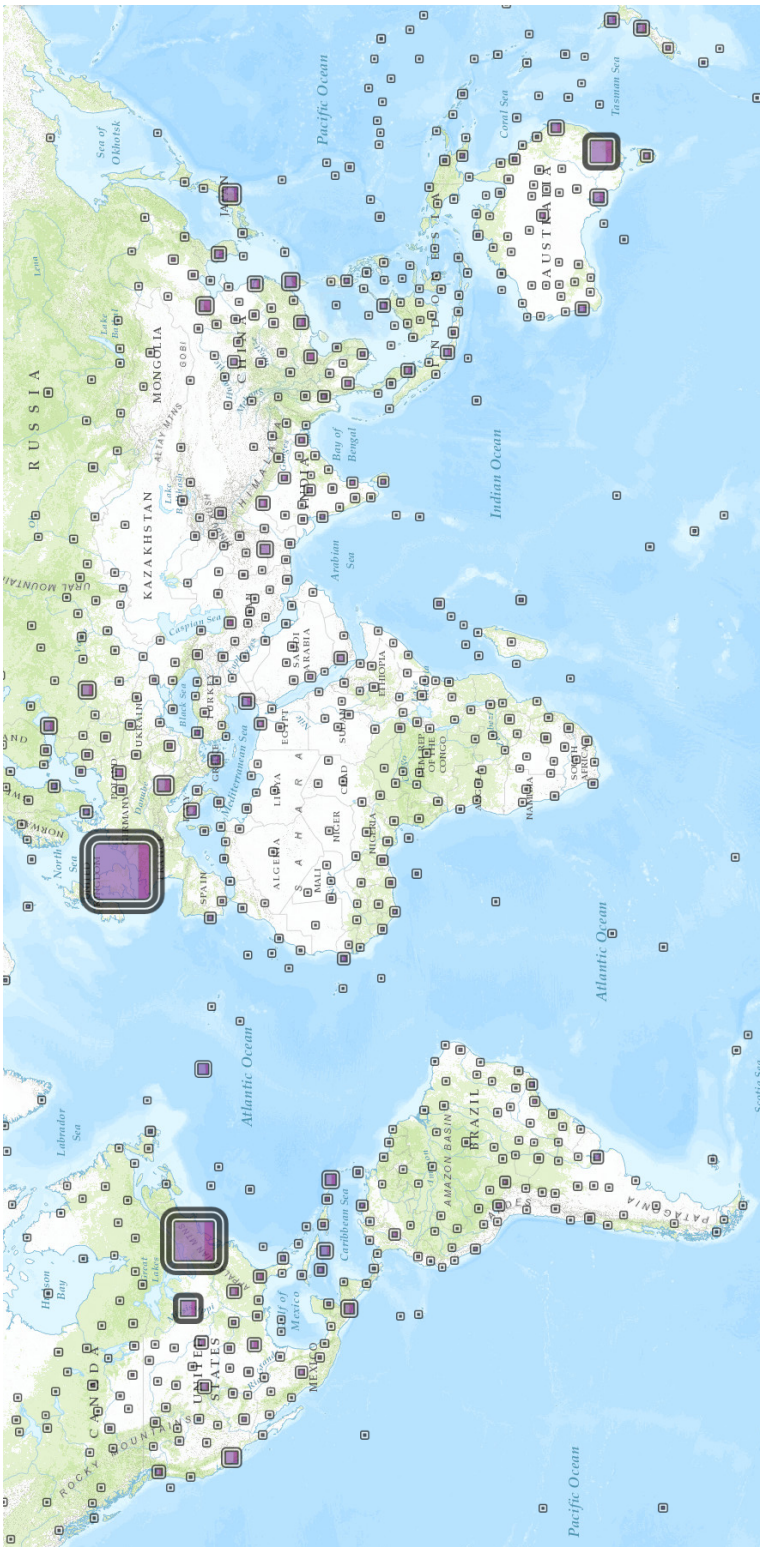


Figure 4.3: This figure shows an overview of 8000 locations in Trove at which the most books were unveiled (95% of the books in Trove).

Chapter 5

Conclusions

Our visualization successfully shows a clustering of approximately 95% of all books in Trove (published at the 8000 most frequent locations of publishing) with interactive zooming and panning. Not all filters can be applied at interactive speeds due to CPU bottlenecks and lack of supporting indices in the database, but the storage architecture is well-suited for GlamMap. We can improve the speed of data access by executing data-queries in parallel and applying distribution if necessary.

Performing geocoding very accurately fully automatically is difficult, but we can identify incorrectly geocoded locations by hand. If we add functionality that enables users to correct mistakes in metadata and mistakes made during geocoding, then the quality of the data will be improve. Through crowdsourcing we can improve the accuracy of location data and other metadata in Trove. The year in which a book was issued is an example of such metadata. The categorisation of books in Trove by the year of publishing can be more discriminating due to artefacts in the data.

Copies of individual books in Trove and which library holds them can still be imported, which opens the opportunity for another visualization of Trove. We can explore other visualisation options tuned for the dataset of Trove, but have shown this architecture is well-suited for GlamMap.

Bibliography

[1] ArcGIS. http://www.arcgis.com/. 9, 22

[2] GlamMap. http://www.glammap.net/. 12, 57

[3] Leaflet. http://www.leafletjs.com/. 9

[4] Trove. http://trove.nla.gov.au/. 2, 31

[5] K. Been, E. Daiches, and C. Yap. Dynamic map labeling. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):773–780, 2006. 4

[6] Arianna Betti, Dirk H. P. Gerrits, Bettina Speckmann, and Hein van den Berg. GlamMap: Visualising library metadata. In *Proceedings of the VALA 17th Biennial Conference and Exhibition*, Melbourne, 2014. http://www.vala.org.au/vala2014-proceedings/. 4, 10

[7] Katy Börner. iScape: A collaborative memory palace for digital library search results. In *Proceedings of the 9th International Conference on Human-Computer Interaction*, pages 1160–1164, 2001. 1

[8] Katy Börner and Chaomei Chen. Visual interfaces to digital libraries: Motivation, utilization, and socio-technical challenges. In Katy Börner and Chaomei Chen, editors, *Visual Interfaces to Digital Libraries*, volume 2539 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2002. 1

[9] Guoray Cai. Geovibe: A visual interface for geographic digital libraries. In Katy Börner and Chaomei Chen, editors, *Visual Interfaces to Digital Libraries*, volume 2539 of *Lecture Notes in Computer Science*, pages 171–187. Springer Berlin Heidelberg, 2002. 4

[10] Timothy M. Chan. Closest-point problems simplified on the RAM. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA '02)*, pages 472–473, 2002. 17

[11] Timothy M. Chan. A dynamic data structure for 3-D convex hulls and 2-D nearest neighbor queries. *Journal of the ACM*, 57(3):16:1–16:15, 2010. 17

[12] Edward Clarkson, Krishna Desai, and James Foley. Resultmaps: Visualization for search interfaces. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1057–1064, 2009. 4

[13] Borden D. Dent. *Cartography: Thematic Map Design*. McGraw-Hill, 6th edition, 2008. 13

[14] DPLA. DPLA by county and state. `http://dp.la/apps/14/`. 4

[15] Jason A. Dykes. Exploring spatial data representation with dynamic graphics. *Computers & Geosciences*, 23(4):345 – 370, 1997. Exploratory Cartograpic Visualisation. 4

[16] Europeana Foundation. Europeana. `http://www.europeana.eu/portal/`. 4

[17] Mark Gahegan. Beyond tools: visual support for the entire process of GIScience. In Jason Dykes, Alan M. Maceachren, and Menno J. Kraak, editors, *Exploring Geovisualization*, pages 83–99. Elsevier Science, Amsterdam, 2005. 4

[18] Diansheng Guo, Jin Chen, Alan M. MacEachren, and Ke Liao. A visualization system for space-time and multivariate patterns (vis-stamp). *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1461–1474, November 2006. 4

[19] S. Hadlak, H. Schulz, and H. Schumann. In situ exploration of large dynamic networks. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2334–2343, 2011. 5

[20] C. Hurter, A. Telea, and O. Ersoy. Moleview: An attribute and structure-based semantic lens for large element-based plots. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2600–2609, 2011. 5

[21] IFLA. Functional requirements for bibliographic records: final report. Technical report, IFLA Study Group on the Functional Requirements for Bibliographic Records, München: KG Saur, 1998. 24

[22] Alan M. Maceachren, Monica Wachowicz, Robert M. Edsall, Daniel Haug, and Raymon Masters. Constructing knowledge from multivariate spatiotemporal data: integrating geographical visualization with knowledge discovery in database methods. *International Journal of Geographical Information Science*, 13:311–334, 1999. 4

[23] Linn Marks, Jeremy A.T. Hussell, Tamara M. McMahon, and Richard E. Luce. Activegraph: A digital library visualization tool. *International Journal on Digital Libraries*, 5(1):57–69, 2005. 4

[24] OCLC. WorldCat. `http://www.worldcat.org/`. 2, 4, 10, 33

[25] Roeland Scheepens, Huub van de Wetering, and Jarke Jack van Wijk. Non-overlapping aggregated multivariate glyphs for moving objects. In *Proceedings of the 7th IEEE Pacific Visualization Symposium*, pages 17–24, 2014. 4, 15

[26] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, VL '96, pages 336–, Washington, DC, USA, 1996. IEEE Computer Society. 4, 7, 12

[27] Ben Shneiderman, David Feldman, Anne Rose, and Xavier Ferré Grau. Visualizing digital library search results with categorical and hierarchical axes. In *Proceedings of the 5th ACM Conference on Digital Libraries (DL '00)*, pages 57–66, New York, NY, USA, 2000. ACM. 1, 4

[28] Terry A. Slocum, Robert B. McMaster, Fritz C. Kessler, and Hugh H. Howard. *Thematic Cartography and Geovisualization*. Prentice Hall, 3rd edition, 2008. 2

[29] Terry A. Slocum, Jr. Robert S. Sluter, Eritz C. Kessler, and Stephen C. Yoder. A qualitative evaluation of MapTime, a program for exploring spatiotemporal point data. *Cartographica*, 39(3):43–68, 2004. 13

[30] C. Vehlow, T. Reinhardt, and D. Weiskopf. Visualizing fuzzy overlapping communities in networks. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2486–2495, 2013.

[31] Matthew O. Ward. Multivariate data glyphs: Principles and practice. In *Handbook of Data Visualization*, Springer Handbooks of Computational Statistics, pages 179–198. Springer Berlin Heidelberg, 2008.

[32] Erik Wilde. Knowledge organization mashups, 2006. 4

[33] Wesley Willett, Jeffrey Heer, and Maneesh Agrawala. Scented widgets: Improving navigation cues with embedded visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1129–1136, 2007. 12

[34] J. Wood, J. Dykes, A. Slingsby, and K. Clarke. Interactive visual exploration of a large spatio-temporal dataset: Reflections on a geovisualization mashup. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1176–1183, Nov 2007.